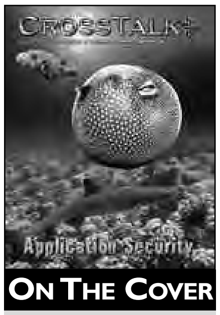
CrossTalk

September 2008 **The Journal of Defense Software Engineering** Vol. 21 No. 9

Application Security



On The Cover

Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

CrossTalk

Co-Sponsors:	
DoD-CIO	*The Honorable John Grimes*
OSD (AT&L)	*Kristen Baldwin*
NAVAIR	*Jeff Schwalb*
76 SMXG	*Daniel Goddard*
309 SMXG	*Karl Rogers*
DHS	*Joe Jarzombek*
Staff:	
Managing Director	*Brent Baxter*
Publisher	*Kasey Thompson*
Managing Editor	*Drew Brown*
Associate Editor	*Chelene Fortier-Lozancich*
Publishing Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
E-mail	stsc.customerservice@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the Department of Defense Chief Information Office (DoD-CIO); the Office of the Secretary of Defense (OSD) Acquisition, Technology and Logistics (AT&L); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). DoD-CIO co-sponsor: Assistant Secretary of Defense (Networks and Information Integration). OSD (AT&L) co-sponsor: Software Engineering and System Assurance. USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG); and Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 18.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole reponsibility of the author and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Smooth Sailing for Application Security

So many of an organization's business practices are now pushed to the Web. In-house and external applications are extending secure data and proprietary information out into a pool of expanded users in order to satisfy business needs. The dangers of this practice include other pool dwellers that might be malicious in nature, seeking to exploit what so many software professionals and system users hold dear – information.

The nature of this issue's focus, *Application Security*, reminds me of the story of Dame Ellen MacArthur, who completed the fastest solo non-stop circumnavigation of the globe in less than 72 days. MacArthur courageously sailed dangerous waters, surrounded by icebergs, massive swells, and sea-dwelling predators similar to the shark on this month's cover (but without the protective capabilities of the porcupine fish). Like MacArthur in her sailboat, making her way through a treacherous sea, precious organizational information flows through application software out to a community of users, vendors, and customers, with malicious hackers always present. This is risky business indeed.

Joe Jarzombek, Director for Software Assurance for the National Cyber Security Division of the Department of Homeland Security (DHS), while addressing an Advanced Software Acquisition Management class at the Defense Acquisition University, held up a copy of the June 2008 issue of CROSSTALK on Software Quality as a useful source and said, "Rather than attempt to break or defeat network or systems security, hackers are opting to target application software to circumvent security controls." The numbers bear that out as Jarzombek noted that Gartner, Inc., an information technology research and advisory company, found that 90 percent of software attacks were aimed at the application layer. Jarzombek also believes that "most exploitable software vulnerabilities are attributable to non-secure coding practices and not identified in testing." He asserted, "Functional correctness must be exhibited even when software is subjected to abnormal and hostile conditions."

So what is a software professional to do? This month's articles provide an application security lifeline. The lead article, *Securing Legacy C Applications Using Dynamic Data Flow Analysis* by Steve Cook, Dr. Calvin Lin, and Walter Chang offers solutions in securing existing code within applications. *Building Secure Systems Using Model-Based Engineering and Architectural Models* by Dr. Jörgen Hansson, Dr. Peter H. Feiler, and John Morley, and *Practical Defense in Depth* by Michael Howard offer structural support to anyone looking to bolster their security practices on future designs. Likewise, Karen Mercedes Goertzel's article, *Enhancing the Software Development Life Cycle* [SDLC] *to Produce Secure Software*, shares the DHS Software Assurance Program's perspective of what makes software secure, while leaving specific SDLC security-development up to the reader. In *Supporting Safe Content-Inspection of Web Traffic*, Dr. Partha Pal and Michael Atighetchi show how Hypertext Transfer Protocol Secure (HTTPS) proxies enable safe interception and inspection of HTTPS traffic, while Corey P. Cunha's article *Hazardous Software Development* explores past safety-critical systems failures and modern software solutions. And finally, this issue's co-sponsor, the DHS Software Assurance Program, offers many free resources for Application Security – just follow the link on the back cover.

So, to all those charged with the difficult duty of securing your organizations applications, remember the promising success of Dame Ellen MacArthur. She, just like you, set sail into dangerous waters, yet she arrived safely and without a single scratch. To all our CROSSTALK readers: I hope this issue helps in navigating through the sea of threats and leads to smooth sailing in all of your security efforts.

Kasey Thompson

Kasey Thompson
Publisher

Securing Legacy C Applications Using Dynamic Data Flow Analysis

Steve Cook
Southwest Research Institute

Dr. Calvin Lin and Walter Chang
University of Texas at Austin

Most attacks on networks and information systems consist of exploits of software vulnerabilities. Even safe programming languages are not immune to problems such as Structured Query Language (SQL) injection, cross-site scripting, or other information flow-related vulnerabilities. While current technological and educational efforts seek to ensure the security of future applications, millions of lines of existing software must be secured as we work to defend our national infrastructure. What is needed is an automatic and scalable method that identifies and traps runtime exploits and that can update existing software as security policies evolve. This article presents ongoing research on an approach to securing legacy C applications, while remaining applicable to future problems and languages.

Most attacks on networks and information systems begin by exploiting a vulnerability in a software application that is resident on a host computer, server, or even an appliance designed to provide network defense. Addressing these vulnerabilities is currently very labor-intensive, requiring constant updates and patches. All of us have become accustomed to receiving software security updates on an almost daily basis for many of our commonly used applications. Terms such as denial of service, phishing, botnets, and spamming are all becoming part of our everyday vernacular, and our collective concern.

Current technological and educational efforts seek to ensure the security of future applications. Most approaches to resolving today's security concerns focus on single-point solutions. Furthermore, millions of lines of existing software that comprise our legacy systems must be secured to defend those information systems on which our national infrastructure depends. Unfortunately, today's software engineers are not typically trained in the development of secure software systems. Implementation of application security requires that the programmer be an expert not only in the application domain, but also in secure coding practices. Our universities are beginning to add security to the software educational curriculum so that new graduates can distinguish good practices from bad as they translate software designs into source code. The Department of Homeland Security has created a publicly available Web site to capture best practices in developing secure software at <https://buildsecurityin.us-cert.gov>. However, even when a programmer is trained in best practices for secure programming, it is unrealistic to depend upon the programmer to develop code that is absent of vulnerabilities.

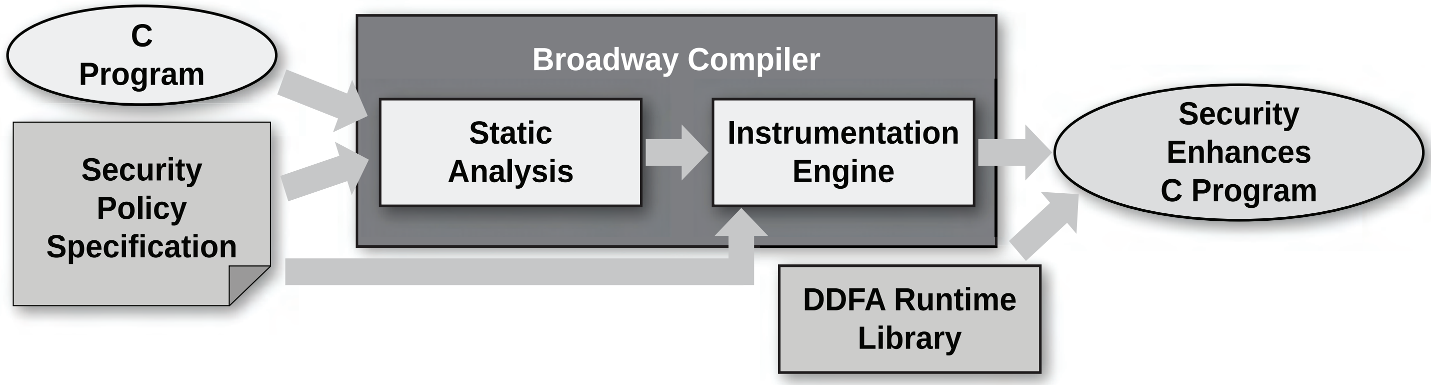
To supplant the ad-hoc security enhancing efforts of today and meet the challenges of tomorrow, an automated method is needed to identify and guard against runtime exploits while remaining flexible and agile as security policies change and evolve. In this article, we present research work in progress to develop a system that ensures that C programs enforce a wide variety of user-defined security policies with a minimum of runtime overhead and disruption to development processes. In the future, our system can be extended to handle multiple languages and complement new security solutions.

What Is Dynamic Data Flow Analysis (DDFA)?

DDFA is an extensible, compiler-based system that automatically instruments the source code of arbitrary (meaning without any assumptions on the code) C programs to enforce a user-specified security policy. The system does not require any modification to the original source code by the developer and also does not significantly degrade a program's runtime performance. Moreover, it has the ability to simultaneously enforce many different classes of security vulnerabilities.

The DDFA system is built upon the Broadway static data flow analysis and error checking system, which is a source-to-source translator for C developed by the computer sciences department at the University of Texas at Austin (UT-Austin) [1]. UT-Austin and the Southwest Research Institute (SwRI) are collaborating to enhance the Broadway specification language and analysis infrastructure with a dependence analysis, an instrumentation

Figure 1: *Overall Architecture of DDFA System*



engine, and a dynamic data flow library.

Figure 1 shows the overall architecture of the DDFA system. Input to the Broadway compiler consists of the source code of an untrusted program and a security policy specification file. The output is an enhanced version of the original source code that has been automatically instrumented with DDFA runtime library calls. The modified program is then compiled for the platform of choice so that its security policy can be enforced at runtime using DDFA. The system does not require hardware or operating system changes.

The primary design goals are the following:

1. Minimize the required developer involvement.
2. Minimize runtime overhead of the secured program.
3. Provide sufficient generality for multi-level security support and enough extensibility for future capabilities.

Minimizing developer involvement is achieved through a security policy specification file that is independent of the program. A security policy is defined once by a security expert using a simple language, which has a direct mapping to the application programming interface to which the program is written. The policy, once defined, can be applied to many different programs. The DDFA approach is easily integrated into the development workflow, adding only an additional compilation step before application deployment.

To minimize the runtime overhead of an executing program, the DDFA approach builds on the body of research in static analysis and leverages semantic information provided by the security policy to enable optimizations beyond standard compiler techniques. This results in a program that is instrumented with additional code only where *provably necessary*, so innocuous flows of data are not tracked at runtime, thus keeping runtime overhead low.

For sufficient generality and extensibility in security exploits, a DDFA approach is used instead of a more traditional dynamic taint analysis [2, 3, 4]. Taint analysis tracks the flow of tainted data (i.e., data originating from the potential attacker) through the system at runtime and then checks that the tainted data is not misused. Our approach expands on the generality of taint tracking by recognizing that taint tracking is a special case of data flow tracking (DDFA). Whereas taint analysis typically tracks one bit of information, data flow analysis can track multiple bits of information and can combine the information in more flexible ways than taint analysis. This allows our approach to support multi-level security and provide a higher level of generality that could be used for other unanticipated security challenges in the future.

Traditional Tainted Data Attacks	Security Problems Taint Tracking Cannot Handle
Format String Attacks	File Disclosure Vulnerabilities
SQL Injection	Labeled Security Enforcement
Command Injection	Role-Based Access Control
Cross-Site Scripting	Mandatory Access Control
Privilege Escalation	Accountable Information Flow

Table 1: *Vulnerabilities That Taint Tracking Can and Cannot Handle*

Table 1 identifies exploits that can and cannot be handled by taint-based systems. Since the DDFA approach supports general data flow tracking, it can handle all of these exploits, and can do so simultaneously. Such generality will become particularly important as developers move to memory-safe languages such as Java and C#, where the use of taint tracking to enforce secure control flow is not as important. Security vulnerabilities that are not currently addressed by the DDFA approach include those such as covert timing channel vulnerabilities, since they are outside the scope of data flow analysis [5].

"To supplant the ad-hoc security-enhancing efforts of today and meet the challenges of tomorrow, an automated method is needed to identify and guard against runtime exploits while remaining flexible and agile as security policies change and evolve."

How Does DDFA Work?

In this section, we will discuss in more detail the components of the DDFA system and how they work. The components include the following: the *policy specification*, *static data flow analysis*, *instrumentation engine*, and *DDFA*.

Policy Specification

The policy specification is defined using the Broadway specification language [1, 6]. It is a simple declarative language that is used to tell the compiler how to perform specific data flow analysis by supplying rules. The two high-level items to specify are the *lattice* definition and the *summaries* for the library functions and system calls.

A lattice must be defined for each type of analysis to be performed. For example, in a format string attack, taint analysis can be used. A lattice called *Taint* would be defined with two flow values which could be named *Tainted* and *Untainted*. Furthermore, *Untainted* is placed higher than *Tainted* to signify that when the two flow values are combined on a particular object being tracked, the result is the lower of the two, *Tainted*. Lattices naturally model hierarchical security levels and are ideal for reasoning about multilevel security and access control that go beyond taint tracking, such as role-based access control (RBAC) [7].

Summaries for the library functions and system calls define how the lattice flow values are introduced into the system, how the flow values propagate through the system, and how to track unsafe use of the lattice flow values. Continuing our format string attack example, the following specification declares that data introduced into the system through the network library call *recv()* would always be tainted:

```
procedure recv(s, buf, len, flags)
{
 on_entry { buf → buffer }
 analyze Taint { buffer ← Tainted }
}
```

Here, the *on_entry* keyword describes function parameters relevant to the analysis and gives a name, *buffer*, to the object pointed to by *buf*. The *analyze* keyword describes the effect of the function on lattice flow values.

To allow the compiler to reason about the propagation of tainted data, the following specification of the library system call *strdup()* declares that it returns a pointer to *string_copy* and that *string_copy* should have whatever taintedness that *string* has:

```
procedure strdup(s)
{
 on_entry { s → string }
 on_exit { return → string_copy }
 analyze Taint { string_copy ← string }
}
```

Finally, to track the unsafe use of tainted data, an error handler would be specified as follows in the summary for a library call such as *printf()* (an ultimate perpetrator in a format string attack) when unsafe data is actually used:

```
procedure printf(format, args)
{
 on_entry { format → format_string }
 error_if (Taint : format_string could-be
           Tainted)
     fsv_error_handler();
}
```

Here, the *error* keyword signals to the compiler that special action is required when the condition is met.

Unlike most taint tracking systems, the security policy (lattice and summary declarations) and underlying analysis is not hard-coded into the compiler or runtime system. Instead, this semantic information for the analysis is provided in a separate specification file. This separation allows the system to remain general and flexible enough to enforce a wide variety of security policies.

Some insight into the generality and flexibility of the DDFA system can be demonstrated by how it guards against *file disclosure* attacks which cannot be handled with traditional taint tracking systems. For the analysis, two lattices must be defined as follows to track the origin of data and its trustedness:

```
property Kind : { File, Filesystem, Client,
                  Server, Pipe, Command,
                  StandardIO,
                  Environment,
                  SystemInfo, NameServer
                  }

property Trust : {Remote, External,
                  Internal }
```

For a true file disclosure problem, only *File* is used, but this definition could be reused for other policies that need to distinguish between other sources. For the sake of brevity, summaries for how lattice flow values are introduced into the system and how those flow values are propagated will not be shown here, as it is somewhat similar to the previous format string attack example. The following specification defines a violation of the policy where *File* data from a file with *Remote* trustedness is sent to a *server* socket with *Remote* trust (indicating that it was initiated by a remote user):

```
procedure write(fd, buf_ptr, size)
{
 on_entry { fd → IOHandle, buf_ptr →
       buffer }
 error if ( (Trust : buffer could-be Remote
   && Kind : buffer could-be File) &&
   (Trust : IOHandle could-be Remote
   && Kind : IOHandle could-be Server) )
     file_disclosure_error_handler()
}
```

Static Data Flow Analysis

To avoid the cost of tracking all objects at runtime, the DDFA system performs inter-procedural data flow analysis that identifies all program locations where policy violations might occur. A subsequent inter-procedural analysis identifies all program statements that affect the lattice flow value of objects that may trigger a policy violation. Both of these compiler passes are supported by a fast and precise pointer analysis for potentially aliased data.

The first pass statically identifies all possible violations of the security policy. If the analysis can prove that there are no such vulnerabilities in the program, no further analysis or instrumentation is needed. When analysis determines that a vulnerability exists or cannot determine if a vulnerability is genuine, additional analysis is needed to determine where instrumentation is required.

The second pass is an inter-procedural dependence analysis that identifies all the statements in the program that affect the flow value of objects that may trigger a security violation. This pass provides a list of statements to the instrumentation engine to be inserted into the original source code so that dynamic data flow analysis can be performed at runtime.

Pointer analysis is necessary for memory-unsafe languages like C and C++ because objects could have many different pointers pointing to them, making it trickier to reason precisely about the flow of data. The limited scalability of pointer analysis has stymied previous attempts to apply inter-procedural analysis to dynamic taint tracking [8]. However, the DDFA system makes use of a highly scalable and accurate client-driven pointer analysis that leverages the semantic information provided by the security policy to dramatically reduce the amount of code that needs to be instrumented [9, 10].

Instrumentation Engine

The instrumentation engine uses the results from static data flow analysis to determine where in the program additional code must be inserted to support the DDFA at runtime. The instrumentation engine also uses the semantics of the security policy specification to determine which particular calls from the DDFA runtime library will be inserted into the program. Continuing our format string attack example, the following C code snippet shows how the network library call *recv()* would be instrumented if static analysis determines that it may introduce suspect data into the system:

```
recv(sock_fd, recv_msg, 10, 0);
ddfa_insert(DDFA_LTAINT, recv_msg,
            strlen(recv_msg),DDFA_
            LTAINT_TAINTED);
```

The *ddfa_insert()* call sets the memory region occupied by the string object *recv_msg* as *TAINTED*. Any attempt to copy the data occupied by this object to another memory region or to use this data in an unsafe manner would pass the flow value tainted to the copied object or trigger the error handler respectively. To support propagation of tainted data, the following code snippet shows how the library call *strdup()* would be instrumented:

```
copy_msg = strdup(recv_msg);
ddfa_copy_flowval(DDFA_LTAINT,
copy_msg, recv_msg, strlen(copy_msg));
```

Finally, the following code snippet shows how the library call *printf()* would be instrumented to conditionally invoke an error handler before unsafe use of tainted data occurs:

```
if ( ddfa_check_flowval(DDFA_LTAINT,
  copy_msg, DDFA_LTAINT_TAINTED) )
  {fsv_error_handler();}
printf(copy_msg);
```

This example illustrates how the DDFA approach can go much further than simply detecting vulnerabilities, as with purely static approaches, by allowing the original program to execute securely even if the programmer does not fix the underlying problem. Additionally, the flow of the original program is not affected unless a potential unsafe use of tainted data is detected.

DDFA

The DDFA is supported by the data flow analysis at runtime library and, for each type of analysis, tracks the lattice flow values for all objects identified by the static analysis as potentially unsafe. As the program executes, our system tracks object flow values at the byte level, which provides fine-grained tracking that is necessary in memory unsafe languages such as C. If an unsafe use of an object occurs, the error handler specified in the security policy will be invoked before the object is actually used in an unsafe manner. The primary benefit of performing data flow analysis at runtime is to subvert a security attack by invoking the appropriate error handler specified in the security policy before an unsafe use of an object occurs.

Effectiveness of the DDFA Approach

In this section we will address, in both objective and subjective terms, the effectiveness of the DDFA approach.

Minimizing Impact on Development Processes

One of the most important benefits in using the DDFA system is that it works on existing C programs and does not require the developer to make any changes to the original source code. The developer need only recompile their program with the DDFA system to create a security-enhanced version of their original source code. If not using one of the default policy specifications provided by our system, such as *taint tracking* or *file disclosure vulnerability*, a security expert can extend the system by creating a new policy specification file. In contrast, to extend a conventional taint tracking system to something other than taint tracking, core components of the compiler infrastructure would have to be rewritten. In the DDFA system, the only change is in the policy specification file itself. This also allows new security policies to be developed quickly in response to new attacks, resulting in a more agile response to the ever-changing security environment.

Although the DDFA approach currently requires the source code of the application as input to the system instead of the binary executable, it does not require the implementation source code of the library functions and system calls that appear as summaries in the policy specification file. Only an understanding of the behavior of the function calls is required. Moreover, the work described in this article is part of an ongoing research program, which has as a longer-term goal of applying the DDFA approach to binary executables. This could be an evolutionary step in the research, since the DDFA system already performs its analysis on a low-level representation of the input program. However, there are challenges in moving to binary because significant information that is leveraged in minimizing the performance impacts of security insertion is lost in the compilation process.

"One of the most important benefits in using the DDFA system is that it works on existing C programs and does not require the developer to make any changes to the original source code."

Another related aspect of our system is that the security enforcement is added to the program after it has been developed. This opens up many new software engineering possibilities such as code separation. In-house software will be more maintainable and agile because it is unfettered by security concerns. Likewise, commercial off-the-shelf and open source software can be brought into a highly trusted environment and then automatically made more secure. It also allows an organization to keep their security policy specification private when subcontracting software development to outside organizations.

Program	Original	DDFA	Runtime Overhead
pfingerd	3.07 seconds (s)	3.19 s	**3.78 %**
muh	11.23 milliseconds (ms)	11.23 ms	**0.0 %**
wu-ftpd	2.745 megabytes per second (MB/s)	2.742 MB/s	**< 1 %**
bind	3.580 ms	3.566 ms	**< 1%**
apache	6.062 MB/s	6.062 MB/s	**< 1%**
		Average Overhead:	**0.65 %**

Table 2: *Runtime Overhead for Server Programs Performing Taint Analysis*

Minimizing Performance Overhead

Another important benefit is that the DDFA approach takes advantage of the fact that only a very small portion of a program is actually involved in any given security attack [11]. Identifying this small portion of the program, however, requires sophisticated static analysis. Without this analysis, large portions of the program would have to be instrumented, substantially increasing the program's runtime overhead.

In order to quantify the runtime overhead that is incurred on programs enhanced by the DDFA system, we measured the runtime overhead for two different sets of *open-source* C programs. In the first set, we measured response time or throughput for several different server type programs with considerable input/output. We took measurements against the original program and then again after instrumentation by the DDFA system (in this case, applying a taint-based policy specification). As shown in Table 2, our solution has an average overhead of 0.65 percent. The current fastest compiler-based and dynamically optimized systems report server application overhead of 3-7 percent, and 6 percent, respectively [12, 13].

In the second test set, we measured Standard Performance Evaluation Corporation (SPEC) workloads of four SPECint 2000 benchmarks that are compute-bound applications after performing a format string vulnerability analysis. In each case, our static analysis determines that the programs contain no such vulnerability, as expected. Thus, the true overhead for these examples is zero percent. In order to understand the impact of our system on compute-bound programs that do contain vulnerabilities, we manually inserted a vulnerability into each of the benchmarks. As can be seen in Table 3 (see next page), the average runtime overhead is less

Program	Runtime Overhead
gzip	51.35 %
vpr	< 1 %
mcf	< 1%
crafty	< 1%
Average:	12.9 %

Table 3: *Runtime Overhead for Compute-Bound Programs Performing Format String Vulnerability*

than 13 percent, which is significantly better than the best previously reported averages of 75-260 percent [12, 13].

Minimizing Code Expansion

Because our system adds instrumentation to the source program, it expands the programs static code size. To quantify this property, we measured code expansion by comparing the sizes of the original and modified binary executables after performing taint tracking on the server programs mentioned previously. As can be seen in Table 4, the average expansion is less than 1 percent. Compiler-based systems such as the GNU Image-Finding Tool report 30-60 percent increases in binary size [8].

Related Capabilities and Future Directions

Finally, another important benefit of the DDFA system is that the technology is applicable to future threats and other areas that are not specific to security. For example, systems that depend on semantic information such as information flow (i.e., privacy concerns) or access controls can be enhanced by our system without modifying the core infrastructure. We recently demonstrated this generality by showing that the policy specification file can be used to define roles in a RBAC system, and then subsequently applied to a set of software that previously did not have RBAC. The DDFA system can go beyond problems such as buffer overflows and overwrite attacks that continue to plague legacy languages and solve problems affecting even safer languages such as Java and C#. These languages are not immune to attacks like SQL injection and cross-site scripting that depend on semantic, not language-level, errors in data handling. DDFA, along with the semantic information captured by the policy specification, can address these types of problems.

Our challenge, therefore, is to develop solutions that can both be applied to existing legacy software today for immediate benefits while also looking forward to the more sophisticated challenges that face us in the future. We believe that the DDFA system is well positioned to provide a practical approach to enhancing the security of legacy software in the near future. We are also continuing our research in increasing the scalability of pointer analysis, integrating language-independence into the DDFA technology, as well as researching and testing the breadth of applicability of the approach itself.◆

Acknowledgements

This work is funded in part by the Intelligence Advanced Research Projects Activity National Intelligence Community Enterprise Cyber Assurance Program. Additionally, the authors would like to acknowledge the following staff who form part of this research team: Ben Hardekopf and Brandon Streiff (of UT-Austin); Mark Brooks, Nakul Jeirath, Arif Kasim, Ronnie Killough, Jeremy Price, and Galen Rasche (all of the SwRI).

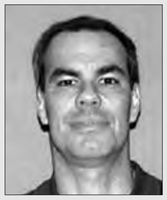
References

1. Guyer, S.Z. "Incorporating Domain-Specific Information into the Compilation Process." Diss., The University of Texas at Austin, Austin, TX, 2003.
2. Newsome, J., and D. Song. Dynamic Taint Analysis for Automation Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Proc. of Network and Distributed Security Symposium, San Diego, CA, 2005.
3. Nguyen-Tong, A., et al. Automatically Hardening Web Applications Using Precise Tainting. Proc. of 20th IFIP International Information Security Conference, 2005.
4. Chen, S., J. Xu, N. Nakka, Z. Kalbarczyk, and R.K. Iyer. Defeating Memory Corruption Attacks Via Pointer Taintedness Detection. Proc. of International Conference on Dependable Systems and Networks, 2005: 378-387.
5. Cabuk S., C. Brodley, and C. Shields. IPCovert Timing Channels: Design and Detection. Proc. of the 11th ACM Conference on Computer and Communications Security, 2004: 178-187.
6. Guyer, S.Z., and C. Lin. An Annotation Language for Optimizing Software Libraries. Proc. of the 2nd Conference on Domain-Specific Languages, 1999: 39-52.
7. Denning, D. "A Lattice Model of Secure Information Flow." Communications of the ACM 19.5 (1976): 236-243.
8. Lam, L.C., and T.C. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. Proc. of the 22nd Annual Computer Security Applications Conference, 2006.
9. Strom, R., and S. Yemini. "Typestate: A Programming Language Concept for Enhancing Software Reliability." IEEE Transactions on Software Engineering 12.1 (1986): 157-171.
10. Guyer, S.Z., and C. Lin. Client-Driven Pointer Analysis. Proc. of the 10th Annual Static Analysis Symposium, June 2003.
11. Newsome, J., D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. Proc. of the Network and Distributed Security Symposium, 2006.
12. Xu, W., S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. Proc. of the 15th USENIX Security Symposium, 2006.
13. Qin, F., C. Wang, Z. Li, H. Seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Information Flow Tracking System for Detecting Security Attacks. Proc. from the 39th Annual IEEE/ACM Symposium on Microarchitecture, 2006: 135-148.

Table 4: *Static Code Expansion After Performing Taint Tracking*

Program	Original (bytes)	DDFA (bytes)	Code Overhead
pfingerd	49,655	49,655	**0.0 %**
muh	59,880	60,488	**1.01 %**
wu-ftpd	205,487	207,997	**1.22 %**
bind	215,669	219,765	**1.90 %**
apache	552,114	554,514	**0.43 %**
		Average Overhead:	**0.91 %**

About the Authors



Steve Cook is a senior research analyst in the System Security and High Reliability Software section at the SwRI. His background and expertise are in parallel and real-time computing, compilers, as well as object-oriented and generic programming. Cook received his master's degree in computer science from Texas A&M University. While at Texas A&M, he worked as a research assistant for Dr. Bjarne Stroustrup, creator of the C++ Programming Language, where he helped develop a new approach to writing concurrent programs free from race conditions and deadlock.

SwRI
System Security and
High Reliability Software
P.O. Drawer 28510
San Antonio, TX 78228-0510
Phone: (210) 522-6322
E-mail: steve.cook@swri.org

Calvin Lin, Ph.D., is an associate professor of computer sciences at UT-Austin and director of their Turing Scholars Honors Program. His research takes a broad view at how languages, compilers, and microarchitecture can improve system performance and programmer productivity. Lin leads the development of the Broadway compiler, which exploits domain-specific information in the compilation process. He holds a doctorate in computer science from the University of Washington.

Department of Computer Sciences
UT-Austin
1 University Station C0500
Austin, TX 78712-1188
Phone: (512) 471-9560
E-mail: lin@cs.utexas.edu

Walter Chang received his bachelor's degree in computer science from Cornell University and is currently a doctoral student in the Department of Computer Sciences at UT-Austin, where his research develops program analyses to improve various aspects of software quality, including software security and software correctness.

Department of Computer Sciences
UT-Austin
1 University Station C0500
Austin, TX 78712-1188
Phone: (512) 232-7434
E-mail: walter@cs.utexas.edu

Web Sites

Department of Homeland Security's (DHS) Software Assurance Program

www.us-cert.gov/swa

The DHS Software Assurance Program spearheads the development of practical guidance and tools and promotes research and development of secure software engineering, examining a range of development issues from new methods that avoid basic programming errors to enterprise systems that remain secure when portions of the system software are compromised. Through collaborative software assurance efforts, stakeholders seek to reduce software vulnerabilities, minimize exploitation, and address ways to improve the routine development and deployment of trustworthy software products.

The Open Web Application Security Project (OWASP)

www.owasp.org

The OWASP is a worldwide free and open community focused on improving the security of application software. Their mission is to make application security "visible" so that people and organizations can make informed decisions about application security risks.

SecurityFocus

www.securityfocus.com

SecurityFocus provides the most comprehensive and trusted source of computer and application security information on the Internet. It provides objective, timely, and comprehensive security information to all members of the security community. Information includes computer security vulnerability announcements, security-related news stories and feature articles, effective security measure implementation ideas, and a high volume, full-disclosure mailing list for detailed discussions.

MILS: High-Assurance Security at Affordable Costs

www.cotsjournalonline.com/home/article.php?id=100423&pg=1

This *COTS Journal* article explores how multiple independent levels of security (MILS) build high-assurance systems that must survive high-threat environments. The central idea behind MILS is to partition a system in such a way that 1) the failure or corruption of any single partition cannot affect any other part of the system or network, and 2) each partition can be security-evaluated and certified separately.

Build Security In (BSI)

http://buildsecurityin.us-cert.gov

BSI contains and links to best practices, tools, guidelines, rules, principles, and other resources that software developers, architects, and security practitioners can use to build security into software in every phase of its development. BSI content is based on the principle that software security is fundamentally a software engineering problem and must be addressed in a systematic way throughout the software development life cycle.

Building Secure Systems Using Model-Based Engineering and Architectural Models

Dr. Jörgen Hansson, Dr. Peter H. Feiler, and John Morley
Software Engineering Institute

The Department of Defense's policy of multi-level security (MLS) has long employed the Bell-LaPadula and Biba approaches for confidentiality and integrity; more recently, the multiple independent levels of security/safety (MILS) approach has been proposed. These approaches allow designers of software-intensive systems to specify security levels and requirements for access to protected data, but they do not enable them to predict runtime behavior. In this article, model-based engineering (MBE) and architectural modeling are shown to be a platform for multi-dimensional, multi-fidelity analysis that is conducive for use with Bell-LaPadula, Biba, and MILS approaches, and enables a system designer to exercise various architectural design options for confidentiality and data integrity prior to system realization. In that way, MBE and architectural modeling can be efficiently used to validate the security of system architectures and, thus, gain confidence in the system design.

System designers face several challenges when specifying security for distributed computing environments or migrating systems to a new execution platform. Business stakeholders impose constraints due to cost, time-to-market requirements, productivity impact, customer satisfaction concerns, and so forth. Thus, a system designer needs to understand requirements regarding the confidentiality and integrity of protected resources (e.g., data). Additionally, a designer needs to predict the effect that security measures will have on other runtime quality attributes such as resource consumption, availability, and real-time performance. After all, the resource costs associated with security can easily overload a system. Nevertheless, security is often studied only in isolation and late in the process. Furthermore, the unanticipated effects of design approaches or changes are discovered only late in the life cycle when they are much more expensive to resolve[1].

MBE for Security Analysis

Modeling of system quality attributes, including security, is often done – when it is done – with low-fidelity software models and disjointed architectural specifications by various engineers using their own specialized notations. These models are typically not maintained or documented throughout the life cycle, making it difficult to obtain a system view. However, a single-source architecture model of the system that is annotated with analysis-specific information allows changes to the architecture to be reflected in the various analysis models with little effort; those models can easily be regenerated from the architecture model (Figure 1). This approach also allows the designer to conduct an adequate trade-off analysis and evaluate architectural variations prior to system realization, thereby gaining confidence in the architectural design. Models also can be used to evaluate the effects of reconfiguration and system revisions in post-development phases.

Using MBE tools, the Software Engineering Institute (SEI) has developed analytical techniques to:

- Represent standard security protocols for enforcing confidentiality and integrity, such as Bell-LaPadula [1, 2], Chinese wall [3, 4], role-based access control [5], and the Biba model [6].
- Model and validate security using system architecture according to flow-based approaches early and often in the life cycle.

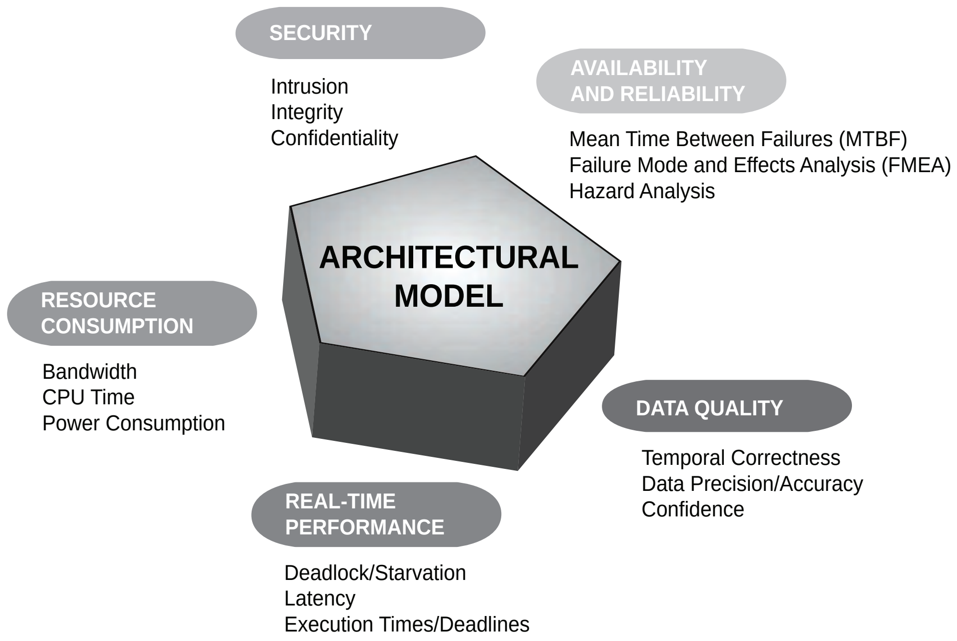
The MBE tools that the SEI uses are the Architecture Analysis and Design Language (AADL) and the Open Source AADL Tool Environment (OSATE) set of analysis plugins [7][2]. The AADL is used to model and document system architecture and provide the following platform for analyses:

- Using a single architecture model to evaluate multiple quality attributes, including security.
- Early and often during system design or when upgrading existing system architecture.
- At different architecture refinement levels as information becomes available.
- Along diverse architectural aspects, such as behavior and throughput.

Architectural Considerations

Security as an architectural concern crosscuts all levels of the system (application, middleware, operating systems, and hardware). Thus, security requires intra- and inter-level validation and has immediate effects on the runtime behavior of the system, specifically on other dependability attributes.

Figure 1: *A Single, System Architectural Model Annotated for Multiple Non-Functional Quality Analyses*



The designer needs to enforce intra- and inter-level security throughout the architecture. Figure 2 depicts various system levels involved in the validation of security privileges against confidentiality requirements (it assumes that authentication and other necessary security services are enforced). The designer seeks to ensure that the software applications do not compromise the confidentiality of the secure information they are exchanging. Consequentially, software applications need to execute on top of a secure operating system, be mapped to a protected and secured hardware memory space, and communicate over a secure communication channel. If the data is labeled "confidential," then every architectural layer needs to have a clearance of at least that level.

Additionally, the designer needs to acknowledge that security comes with a cost. Encryption, authentication, security, and protection mechanisms increase bandwidth demand in terms of the central processing unit (CPU), the network, and memory. These increases affect the temporal behavior of the system (worst-case execution time, response time, schedulability, and end-to-end latency) as well as power consumption (especially important in battery-driven or limited lifetime devices such as sensor networks or portable communication devices).

As a result, security cannot be considered in isolation. The system designer makes choices to trade these quality attributes against each other (a particular concern for embedded and real-time systems, which operate under significant resource constraints while ensuring high levels of dependability and security). Security is interlinked with other non-functional behaviors such as predictability/timeliness and resource consumption, as well as inadvertent effects on reliability and availability. Figure 3 illustrates some of those dependencies on the single-model, multiple-analysis view.

An MBE Approach to Validating Confidentiality

Confidentiality addresses concerns that sensitive data should be disclosed to or accessed only by authorized users (i.e., enforcing prevention of unauthorized disclosure of information). Data integrity is closely related, as it concerns prevention of unauthorized modifications of data.

To model and validate the confidentiality of a system, we distinguish between general and application-dependent validation. General validation of confidentiality is the process of ensuring that a modeled system conforms to a set of common conditions that support system confidentiality independent of a specific reasoning framework for security. MBE takes advantage of the versatile concept of subjects operating on objects by permissible access (read, execute, append, and write), a notion introduced by Bell and LaPadula [1], enabling us to model and validate security at both the software and hardware levels.

This form of validation assumes that subjects and objects are assigned a security level that is the minimum representation to enforce basic confidentiality and need-to-know principles. By contrast, application-specific validation relies on detailed confidentiality requirements and a specific, reasoning-based security framework.

The MBE security framework features:

- Representation of the confidentiality requirements of resources (i.e., objects).
- Representation and generation of security clearance/least privileges[3] of subjects operating on the objects.
- Representation of authorized operations, ensuring unauthorized infiltration, unauthorized exfiltration, and unauthorized median of actions. This is captured in an access matrix.

With the object's security requirements specified in an AADL model, the least amount of privileges for the subjects can be generated in a straightforward manner. Given that the subjects' privileges are specified, a mismatch between the least privilege and what has been specified

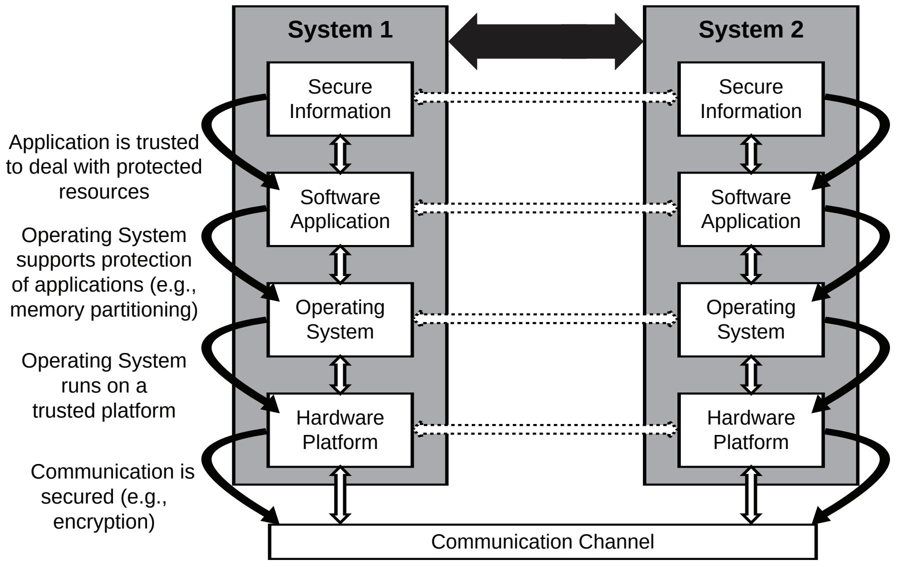
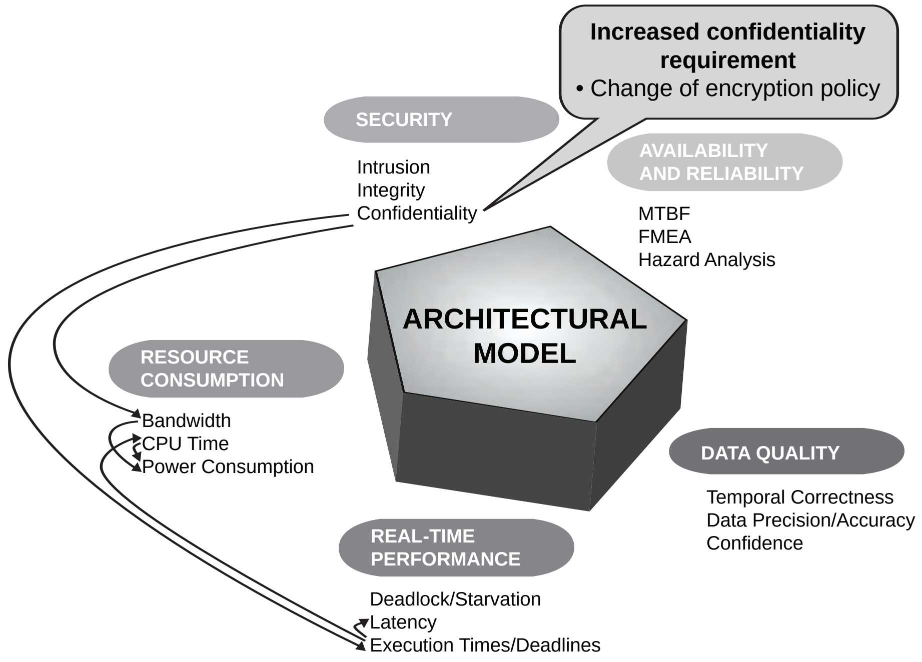

System 1
System 2
Secure Information
Software Application
Operating System
Hardware Platform
Application is trusted to deal with protected resources
Operating System supports protection of applications (e.g., memory partitioning)
Operating System runs on a trusted platform
Communication is secured (e.g., encryption)
Communication Channel

Figure 2: *System Perspective on Security*

Figure 3: *Single Architectural Model Showing an Example of Impact on and Interaction of Non-Functional Behavior Due to a Change in Security*



```
-- Property intended to be customized by modelers.
-- Parameterizes the security property definitions.
property set Security_Types is
   -- Military levels by default
   Classifications:
     Type enumeration (unclassified, confidential, secret,
                             top_secret);
   -- This must be the first element of Classifications
   Default_Classification:
     constant Security_Types::Classifications =>

   -- Default set of categories
   Categories:
      type enumeration (A, B, C, D);
end Security_Types;
```

Figure 4: *Specification of Security Levels*

means the assigned privilege is either insufficient or greater than the minimum privilege. The latter result may be unnecessary or an indication that the subject might be associated with objects not yet described in the model.

The following types of security validation and analysis are available as OSATE plugins:

- **Basic confidentiality principle.** Access should only be granted if given the appropriate security clearance.
- **Need-to-know principle.** Access should be granted to a resource only if there is a need.
- **Controlled sanitization.** Lowering the security level of an object or subject should only be authorized and performed by a privileged subject.
- **Non-alteration of object's security requirements.** A subject using an object as input should not alter the security level of the object, even if the object is updated as an output from the subject.
- **Hierarchical conditions.** A component has (1) a security level that is the maximum of the security levels of its subcomponents, and (2) all connections are checked to determine whether the source component of a connection declaration has a security level that is the same or lower than that of the destination component.

Using OSATE and the AADL, system designers and developers can add analysis techniques as needed.

The validation through architectural modeling of system security – given the confidentiality requirements of data objects and the security clearance by users – must include validation of (1) software architecture, and (2) system architecture where the software architecture is mapped to hardware components.

By mapping the entities of a software architecture (e.g., processes, threads, and partitions) to a hardware architecture (consisting of, for example, CPUs, communication channels, and memory), we can ensure that the hardware architecture supports required security levels, as described in Figures 4 and 5.

Consider the scenario of two communicating processes, both requiring a high level of security because the data objects require secret clearance. The system platform in this scenario consists of a set of CPUs with hardware support for various algorithms that encrypt messages before network transmission. By modeling the system, we can represent and validate that both processes and threads (now considered to be objects) can be executed (access mode) on CPUs (subjects) with adequate encryption support. Furthermore, we can validate that CPUs (objects) communicate data (access modes of writing and reading) over appropriately secured communication channels (subjects). In a similar fashion, we can enforce design philosophies saying that only processes of the same security level are allowed to co-exist within the same CPU or partition, or that they can write to a secured memory.

A combination of the AADL and the OSATE security plugin tool has been put into use in industry. Rockwell-Collins used the technology to enable the high-assurance handling of data from multiple sensors having varying levels of security, such as airborne imagery with the Field Programmable Gate Array (FPGA). Typically, a high-assurance processor is used to securely tag variable input. An FPGA is powerful and fast. It is deemed easier to develop applications on an FPGA, which also reduces the cost and time-to-market. Furthermore, the FPGA can be reprogrammed at runtime (e.g., to fix bugs), which can lower maintenance-engineering costs. Because FPGA behavior is more complex, architecture-level definition and analysis are needed. To this end, Rockwell-Collins developed architectural models of the FPGA using AADL and used the OSATE tool to validate security and demonstrate the high-assurance potential of FPGAs.

Figure 5: *Architectural Components to Which Security Levels and Requirements Can Be Connected*

```
property set Security_Attributes is
  Class: inherit Security_Types::Classifications =>
    value(Security_Types::Default_Classification)
    applies to (data, subprogram, thread, thread group,
                process, memory, processor, bus, device,
                system, port, server subprogram,
                parameter, port group);

  Category: inherit list of Security_Types::Categories =>
    ()
    applies to (data, subprogram, thread, thread group,
                process, memory, processor, bus, device,
                system, port, server subprogram,
                parameter, port group);
  -- . . .
end Security_Attributes;
```

Validating MILS Architectures With the MBE Approach

The AADL and OSATE tools can be used to validate the security of systems designed using the MILS[4] architecture approach (see [8, 9]). MILS uses two mechanisms to modularize – or divide and conquer – in architecting secure systems: partitions, and separation into layers. The MILS architecture isolates processes in partitions that define a collection of data objects, code, and system resources and can be evaluated separately. Each partition is divided into the following three layers, each of which is responsible for its own security domain and nothing else:

1. **Separation Kernel (SK).** Responsible for enforcing data isolation, control of information flow, periods processing,

and damage limitation. An example is the SK Protection Profile [8].
2. **Middleware service layer.**
3. **Application layer.**

Thus, the MILS separates security mechanisms and concerns into the following three component types, classified by how they process data:

- **Single-Level Secure (SLS).** Processes data at one security level.
- **Multiple Single-Level Secure (MSLS).** Processes data at multiple levels, but maintains separations between classes of data.
- **MLS.** Processes data at multiple levels simultaneously and transforms data from one level to another.

The strength of the MILS architecture lies in its reductionist approach to decompose a system into components of the above-mentioned types that would be more manageable to certify. These components are also mapped to partitions (and, as mentioned earlier, the MILS architecture approach builds on partitioning as one key concept to enforce damage limitation and separation of time and space).

An MBE approach is conducive to the validation concerns most critical to MILS, including:

- **Validating the structural rigidity of architecture, such as the enforcement of legal architectural refinement patterns of a security component into SLS, MSLS, and MLS types.** Given that an MILS architecture design and system is decomposed into security components that can be certified in isolation, the structural rigidity concerns the legal mappings and connections of the components. The decomposition into SLS, MSLS, and MLS types can be applied to components, connectors, and ports. Furthermore, each component can be divided into parts using the product, cascade, or feedback decomposition patterns [10, 11, 12]. For example, an MSLS component with *n* security levels can be decomposed into *n* distinct SLS components. Thus, confidence in the validation of an architecture increases with the fidelity of the modeling. By using an architectural model in AADL to capture the security types and multiple architectural levels, MBE analysis is conducted to validate the correctness of the decompositions and mappings.
- **Architectural modeling and validation of assumptions underlying MILS.** Fundamental to enforcement of security in an MILS architecture is having a system that supports partitioning, specifically damage limitation and separation in time and space. By partitioning the system, one minimizes the risk of illegal component interactions among components and protects components from the faulty behavior. This can be realized in the system architecture by ensuring fault-containment and deploying security-cognizant memory allocation so that MILS components and tasks reside in protected memory spaces – and do not co-reside in the same memory space if they differ in security levels. Similarly, separation in time can be ensured through avoiding the interleaved execution of tasks with different security levels, realized in partition scheduling and validating execution behaviors. The AADL supports the modeling of partitions and virtual processors. As well, the virtual machine mechanism is recognized as a key concept for providing robustness through fault containment because it provides time and space partitioning to isolate application components and subsystems from affecting each other (due to sharing of resources). This architecture pattern can be found in the Aeronautical Radio Incorporated (ARINC) 653 standard [13]. A single-source architectural model in AADL can thus be used to validate the security requirement in an architectural context, specifically the MILS composition, and the architectural assumptions required.
- **Validating requirements specific to the NEAT characteristics and the communication system.** MILS requires that its SK and the trusted components of middleware services are implemented so that the security capabilities enforce what is commonly referred to as the NEAT characteristics:
    - N*on-bypassable.* Security functions cannot be circumvented.
    - E*valuatable.* The size and complexity of the security functions allow them to be verified and evaluated.
    - A*lways* invoked. Security functions are invoked each and every time without exception.
    - T*amperproof.* Subversive code cannot alter the function of the security functions by exhausting resources, overrunning buffers, or other forms of making the security software fail.

The MBE approach allows designers to assure that software applications execute on top of a secure operating system, map to a protected and secured hardware memory space, and communicate over secure communication channels. It also enables the analysis of security measures early and throughout the development life cycle.

Conclusions

The objective of a secure system implies that security clearances are given conservatively. The MBE approach supports this objective through enabling analysis of the architectural model to derive the minimum security clearance on components. By providing mechanisms to ensure that sanitization is conducted within allowed boundaries, the MBE approach enables the system designer to analyze and trace more threatening security risks, since sanitizing actions are permitted exemptions of security criteria and rules, and as such should be minimized in the system.

Security analysis using the MBE approach also supports:

- The evaluation of an architecture configuration with respect to impact on other non-functional attributes, such as increases in power consumption, bandwidth usage, and performance.
- The validation of architectural requirements necessary to enforce the MILS approach to containing faults, through partitioning and separation in time and space.
- A reduction of the effort necessary for re-certification in the event of architectural changes.

Furthermore, validation of security can be conducted at multiple layers and different levels of fidelity, early and throughout the development life cycle.◆

References

1. Bell, D.E., and L.J. LaPadula. Secure Computer Systems: Mathematical Foundations MITRE Technical Report 2547, Vol. 1. Bedford, MA: MITRE Corporation, 1973 <www.albany.edu/acc/courses/ia/classics/belllapadula1.pdf>.
2. Bell, D.E., and L.J. LaPadula. Secure Computer Systems: Unified Exposition and MULTICs Interpretation MITRE Technical Report ESD-TR-75-306. Bedford, MA: MITRE Corporation, 1976 <http://csrc.nist.gov/publications/history/bell76.pdf>.
3. Brewer, David D.C., and J. Michael Nash. "The Chinese Wall Security Policy." IEEE Symposium on Security and Privacy. Oakland, CA: 1-3 May 1989 <www.gammassl.co.uk/topics/chinesewall.html>.
4. Lin, T.Y. Chinese Wall Security Policy – An Aggressive Model. Proc. of the Fifth Aerospace Computer Security Application Conference. Tucson, AZ: 4-8 Dec. 1989 <http://ieeexplore.ieee.org/iel5/7100/19131/00884701.pdf>.

5. Ferraiolo, David, and Rick Kuhn. Role-Based Access Control. Proc. of the 15th National Computer Security Conference. Baltimore, MD: 13-16 Oct. 1992 <http://csrc.nist.gov/rbac/ferraiolo-kuhn-92.pdf>.
6. Biba, K.J. Integrity Considerations for Secure Computer Systems MITRE Technical Report-3153. Bedford, MA: MITRE Corporation, Apr. 1977.
7. AADL. Find and Solve Problems Before Runtime With Model-Based Engineering. 14 Apr. 2008 <www.aadl.info>.
8. The Common Criteria Evaluation and Validation Scheme. Validation Protection Profile – U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Vers. 1.03. 11 July 2008 <www.niap-ccevs.org/cc-scheme/pp/id/pp_skpp_hr_v1.03>.
9. Alves-Foss, J., W.S. Harrison, P. Oman, and C. Taylor. "The MILS Architecture for High-Assurance Embedded Systems." International Journal of Embedded Systems 2.3/4 (2006): 239-347.
10. Zhou, J., and J. Alves-Foss. "Security Policy Refinement and Enforcement for the Design of Multi-Level Secure Systems." Journal of Computer Security 16.2 (2008): 107-131.
11. McLean, J. "Security Models." Encyclopedia of Software Engineering. John Wiley & Sons, New York, NY: 1994.
12. Zakinthinos, A. "On the Composition of Security Properties." Diss. U of Toronto, Mar. 1996.
13. ARINC Incorporated. Avionics Application Software Standard Interface, ARINC 653 Standard Document 14 Apr. 2008 <www.arinc.com>.
14. National Institute of Standards and Technology (NIST). The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report. May 2002 <www.nist.gov/director/prog-ofc/report02-3.pdf>.

Notes

1. A NIST study observed that 70 percent of all defects are introduced prior to implementation. Yet only 3.5 percent of the defects were detected in these phases, while 50.5 percent of the faults were detected in the integration phase. The defect removal cost ranged from 5 to 30 times relative to the cost of removing the defect in the phase of introduction (if it had been detected). Other sources are reporting similar estimates; while the numbers vary, the conclusions do not [14].
2. The AADL, an international industry standard, incorporates an XML/XMI exchange format to support model interchange and tool chaining. AADL also can be used (1) with UML state and process charts through its UML profile, (2) to drill into root causes and develop quantitative analysis as a follow-up to the SEI Architecture Tradeoff Analysis Method®, and (3) in conjunction with assurance cases, to support claims made about the safety, security, or reliability of a system. The freely available OSATE includes analysis plugins for performance, resource consumption, security, and reliability.
3. The principle of least privilege has been identified as important for meeting integrity objectives; it requires that a user (subject) be given no more privilege than necessary to perform a job. This principle includes identifying what the subject's job requires and restricting the subject's ability by granting the minimum set of privileges required.
4. MILS has been proposed as an approach to building secure systems [9, 10]. MILS is a joint research effort of academia, industry, and government, led by the U.S. Air Force Research Laboratory. The MILS approach is based on the notion of separating – and thus limiting the scope and reducing the complexity of – the security mechanisms.

® The Architecture Tradeoff Analysis Method is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

About the Authors

Jörgen Hansson, Ph.D., is a senior member of the technical staff and lead of the Performance-Critical Systems (PCS) Initiative at the SEI, and is a professor of computer science at Linköping University, Sweden. The Initiative focuses on developing, maturing, and transitioning analysis-based assurance and model-based engineering tools and practices for predicting the dependability and performance of software systems. Hansson holds bachelor's and master's degrees in computer science from University of Skövde, Sweden, and a doctorate in computer science from Linköping University, Sweden.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213
Phone: (412) 268-6733
Fax: (412) 268-5758
E-mail: hansson@sei.cmu.edu

Peter H. Feiler, Ph.D., is a senior member of the technical staff in the PCS Initiative at the SEI. He has authored more than 80 articles in the areas of dependable real-time systems, architecture languages for embedded systems, and predictable system analysis and engineering. Feiler is the technical lead and author of the SAE AS-2C AADL standard. Feiler holds a Vordiplom degree in math/computer science from Technical University Münich, and a doctorate in computer science from Carnegie Mellon University.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213
Phone: (412) 268-7790
Fax: (412) 268-5758
E-mail: phf@sei.cmu.edu

John Morley is a member of the operating staff at the SEI. He has reported on model-based engineering and service-oriented architecture for SEI publications, has more than 20 years experience in writing and editing scientific and technical materials, and holds a master's degree in English literature from Duquesne University.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213
Phone: (412) 268-6599
Fax: (412) 268-5758
E-mail: jmorley@sei.cmu.edu

Practical Defense in Depth

Michael Howard
Microsoft Corp

As part of its ongoing commitment to Bill Gates' vision of Trustworthy Computing, Microsoft officially adopted important security- and privacy-related disciplines to its software development process. These changes, called the Security Development Lifecycle (SDL) have led to a demonstrable reduction in security vulnerabilities in products such as Microsoft's Windows Vista operating system and its SQL Server 2005 database. The purpose of this article is not to describe the SDL in detail, but to outline some of the practical defensive measurements in use at Microsoft required by the SDL. If Microsoft's SDL is new to you, refer to the page 16 sidebar, "A Brief SDL Overview."

Even though the vulnerability counts have dropped, the number of vulnerabilities is not zero. And, even in my wildest dreams, I do not think we will get to zero. I will explain why shortly.

In the very early days of the SDL, Microsoft focused heavily on removing design and code-level security vulnerabilities; as we progressed, we added processes that help reduce the chance that new vulnerabilities get added to the software.

Examples of implementation requirements in the SDL include:

- Use of code analysis tools on developer's desktops to find security vulnerabilities.
- Removing known insecure functions (such as the C runtime *strcpy* and *strncpy* functions).
- Migrating weak cryptographic algorithms to more robust algorithms (such as Data Encryption Standard to Advanced Encryption Standard, Secure Hash Algorithm (SHA)-1 to SHA-256).

But the SDL is constantly evolving. We update the SDL roughly twice a year so we can keep pace with new vulnerabilities and new security research. We find the continuous improvement to the SDL to be a significant benefit compared to static security certification programs – such as the Common Criteria – which do not evolve so quickly.

Over the last few years, the SDL has been extended to embrace a stronger focus on defense in depth. While some updates to the SDL continue to address design and implementation vulnerabilities, more of today's SDL requirements focus on defense in depth.

The prime driver for this change is a realization that you can never remove all security vulnerabilities from software. The reason this statement is true is simple. Some software you create might be completely secure today, but that could all change tomorrow when security researchers learn (and potentially make public) new classes of vulnerabilities.

Allow me to illustrate the point with a real example.

In October 2003, Microsoft issued a security bulletin, MS03-047 [1] that fixed a cross-site scripting (XSS) vulnerability in the Outlook Web Access (OWA) front end to Microsoft's Exchange 5.5 software. In August 2004, Microsoft issued another bulletin, MS04-026 [2], in the same OWA component that was fixed in MS03-047 to fix an XSS variation called HTTP response splitting. Interestingly, the code fixes were relatively close to one another. So the question that's probably on your mind is, "What happened? How did you guys miss the bug that led to MS04-026?" The answer is simple. At the time we issued MS03-047, the world had not heard of HTTP response splitting vulnerabilities. In theory, the Microsoft Exchange engineers could have scoured the code and fixed every known security bug, but they would probably have missed the vulnerability that led to MS04-026 because nobody knew of that class of vulnerability at the time. So what changed? What led to the security vulnerability? In March 2004, Sanctum (purchased by Watchfire, which has since been purchased by IBM) released a paper entitled "Divide and Conquer" [3] describing a variation of the XSS vulnerability. When the Microsoft engineers fixed the first bug, the second class of bug was unheard of.

"Some software you create might be completely secure today, but that could all change tomorrow when security researchers learn (and potentially make public) new classes of vulnerabilities."

Unfortunately, there were no defense in depth mechanisms in place to protect customers from either of these vulnerabilities, so customers had to apply a security update to protect themselves.

Another example is the integer arithmetic vulnerability [4]. Without going into a detailed explanation, this kind of bug was unheard of 10 years ago, and is a very common security vulnerability today.

The moral of this story is that you can never create totally secure code. There are many reasons why:

- People make mistakes.
- Tools are not perfect.
- Security can have very subtle nuances that only security experts understand, especially with regard to cryptography.
- The Internet is an asymmetric battleground. There are many hidden and skilled attackers who can strike at will, but defenders must be constantly vigilant and never make mistakes.
- Most importantly, we cannot predict future classes of vulnerabilities.

There are two final points that I really wish to stress because these make defense in depth especially critical today:

1. As the security vulnerability landscape evolves, more vulnerability information is moving underground and being used by criminals to attack sensitive systems. The defenders do not know the system is vulnerable, so no security update is available. This is clearly a risk for government systems.
2. Somewhat similar to the first point is that we are seeing more zero-day vulnerabilities. There may be no attacks yet, but there is no update or workaround either.

The rest of this article outlines some

A Brief SDL Overview

The SDL is a set of requirements and recommendations added to an existing software development process to improve security. A side benefit of the SDL is increased robustness since many security vulnerabilities also affect robustness. SDL is all about security improvement, not perfection.

There are two goals to the SDL. The first is to reduce the number of security vulnerabilities in Microsoft products. This is done by removing vulnerabilities from software, or better yet, not adding vulnerabilities to the code from the outset. This reduction in vulnerabilities is achieved through education, tooling, better libraries, and so forth. The second goal is to reduce the severity of any vulnerabilities that are inadvertently left in the product. You can reduce severity by adding defensive mechanisms. The purpose of this article is to explain some of those defenses.

In a nutshell, we teach people to: *Do everything possible to make your product as secure as possible, but assume it will fail.*

A requirement defines an SDL task that must be completed prior to giving the code to customers, and a recommendation defines a best practice that should be considered by the development team. It is not uncommon for a security best practice to start out as a recommendation and then progress to a requirement.

The SDL defines requirements and recommendations for:

- Education.
- Risk assessment.
- Threat modeling.
- Coding.
- Testing.
- Final security review.
- Maintenance.

You can learn more about the SDL in "The Security Development Lifecycle" (Microsoft Press, Howard and Lipner), or at the SDL blog: <http://blogs.msdn.com/sdl>.

of the defense in depth techniques and technologies applied as part of the SDL at Microsoft.

Classes of Defense in Depth Mechanisms

Under the SDL, there are two distinct types of defense in depth mechanisms. We don't call them out explicitly as different classes, but the distinction between the two classes is understood.

The first type of defense is designed to totally stop an attacker from accessing a system or software. This class of defense offers a level of assurance that stops an attacker when the defense is correctly configured and used. If it does not stop an attacker, then the defense has a vulnerability that must be fixed. One example is the ubiquitous firewall. The SDL sets strict requirements on the process for opening an inbound port on the Windows Firewall. Other examples include permissions on objects: a weak permission could render a system insecure. For example, Microsoft issued a security bulletin, MS04-005 [5], for VirtualPC for the Apple Macintosh because of a weak permission on a critical file that led to a symlink-style attack [4].

Strong operating system access controls are an important part of the SDL and critically important to any system; however, a new requirement in the SDL this past year is strong access control mechanisms on database objects. Structured Query Language (SQL) injection vulnerabilities are a well understood vulnerability that can lead to disclosure of sensitive data, data corruption, and, in some cases, system compromise [4]. The industry as a whole has created best practice documentation and tools to help remove these critical bugs, but all it takes is a new attack type or an error to leave a customer open to attack and compromise. The SDL-required defense in depth mechanisms help to mitigate this risk. In short, direct access to the underlying tables is explicitly denied to all but the database administrators, and untrusted access is limited to appropriate database objects such as stored procedures and views. These objects are granted access to the underlying data. This configuration has the effect that if an attacker can bypass the normal defenses against SQL injection, he or she still cannot read the underlying data in the tables. The correct remedy to prevent SQL injection is to build safe SQL queries, but the table-level defense is there solely in case the remedy fails or is implemented incorrectly.

The second type of defense is a set of mechanisms that is designed to slow an attacker down or make an attacker create a different exploit to attack a system. I want to spend most of this article on this subject. At Microsoft, we continue to spend a great deal of time and effort researching, designing, and implementing these defenses: most of them are intended to help mitigate buffer overrun and integer overflow vulnerabilities. A great deal of C and C++ code exists today, and even more is written every day. In a perfect world, people would simply abandon C and C++ in favor of safer programming languages such as C# or Java, but in our imperfect world, C and C++ are often the correct tools for the job. Again, in a perfect world, C and C++ developers would write secure code, but in our imperfect world this is not always possible; however, it is important that C and C++ developers take advantage of defense in depth mechanisms. The SDL mandates a number of important C and C++ defenses, including:

- Address randomization.
- Stack-based buffer overrun detection.
- Heap corruption detection.
- Pointer protection.
- No-execute (often called NX or W^X).
- Service failure restart policy.

Note that none of these defenses actually remove vulnerabilities, nor do they magically make software more secure. What they do is turn a potential code execution exploit into a denial-of-service bug because these defenses will simply fail the application if they detect an anomalous condition. If an application crashes, it also gives the attacker fewer opportunities to re-attempt an attack.

I do not intend to cover each of these in deep technical detail. The reader is urged to e-mail the author or refer to the references for further information [6, 7].

Address Randomization

There is nothing attackers love more than a predictable system since it makes building reliable exploits easier. Reliable exploits are harder to detect because they usually execute correctly and do not crash or alert the system operators to nefarious acts. Windows Vista and Windows Server 2008 (and later) offer image randomization, stack randomization, and heap randomization. Image randomization relocates the entire operating system into one of 256 possible configurations on each reboot. By default, non-operating system images are not randomized, and third-party components must opt into image randomization using the /DYNAMICBASE linker option in Visual C++ 2005 Service Pack 1 and later. Some versions of GNU's C Compiler (GCC) and some versions of Linux and Berkeley Software Distribution (BSD)-based systems also support image randomization by using the -pie compiler option.

Windows Vista and Windows Server

2008 and later also support stack randomization; when a thread is started, the thread's stack is offset by up to 32 pages (4 kilobytes on a 32-bit central processing unit). Again, this option is available by linking with /DYNAMICBASE.

Finally, Windows Vista and Windows Server 2008 (and later) support heap randomization, meaning that when an application allocates dynamic memory from the system heap, the operating system offsets the start of the heap by a random amount. Heap randomization is enabled by default in Windows Vista and Windows Server 2008.

Together, image, stack, and heap randomization can seriously hinder an attacker dealing with the lack of predictability. Usually the sign of a failed attack is a crash in the application under attack. This means that system administrators should really pay attention to applications that crash, as crashes may not just be the sign of a coding bug but a sign of a security bug under attack.

Stack-Based Buffer Overrun Detection

Stack-based buffer overrun detection is available in Microsoft Visual C++ 2003 and later through the /GS compiler switch. It works by adding a random number into a function's stack frame at call time and when the function returns code inserted by the compiler, it verifies that the random number has not changed. If it has changed, then the application crashes because a stack-based buffer overrun has been detected. At this point, we can no longer trust the integrity of the data or the application. Some versions of the GCC offer a similar defense by using -fstack-protector.

This defense could require the attacker to build a specific attack to circumvent the defense. A good example of how this helped protect customer is the Blaster Worm. On Windows Server 2003 (but not Windows 2000 or Windows XP), the vulnerable component was compiled with /GS. The malicious Blaster payload was not aware of the /GS defense, so it crashed Windows Server 2003 machines rather than infecting them with the Blaster Worm.

Heap Corruption Detection

Heap corruption detection is an operating system defense available in Windows Vista and Windows Server 2008 (and later). It is similar in principle to stack-based buffer overrun detection but detects heap metadata corruption. Again, if the heap is corrupted, the operating system can shut down the application, reducing the chance that an attacker will re-attempt a failed attack.

Pointer Protection

C and C++ pointers are an attack vector; if an attacker can overwrite a long-lived pointer in memory, he or she can potentially compromise a computer by writing arbitrary data at a predictable location. Windows includes functions that encode (XOR) a pointer with a random value, and this operation must be reversed successfully in order to get the valid pointer value. In other words, if an attacker attempts to overwrite a pointer, he or she must overwrite it with a value that survives the unencoding operation. Clearly, this is not impossible, but it is another defense the attacker must overcome. The application programming interfaces in Windows that perform these operations are:

- EncodePointer and DecodePointer.
- EncodeSystemPointer and Decode SystemPointer.

Note that GLIBC v2.5 (and later) have a similar defense, but it's mainly used inside GLIBC itself to protect *setjmp* pointers. The functions, defined in *sysdep.h* are PTR_MANGLE and PTR_DEMANGLE.

No-Execute (Often Called W^X)

Microsoft calls no-execute data execution prevention (DEP). This defense marks pages of memory as *Writeable* or *Executable*, but not both. Essentially, this makes it very hard for an attacker to run malicious code out of a writeable memory segment. Most CPUs today support this capability. It is by no means a perfect defense and DEP requires randomization to be effective at stopping a class of attacks known as *return-to-libc* [8].

In Windows, you can link with /NXCOMPAT to opt-in for DEP.

Some versions of BSD and Linux support W^X also, but the compiler and operating system support is not consistent across platforms.

Service Failure Restart Policy

A final defense in Windows Vista and Windows Server 2008 was very hard to implement without sacrificing reliability. Windows uses many services, akin to Unix daemons, to perform critical system tasks. Services are usually long-lived processes that start when a system starts. In many cases, administrators want a crashed service to simply restart. This policy gives better uptime to customers. This is great for reliability, but it can be terrible for security because it means that an attacker can keep trying his attacks over and over until they succeed. The ability to retry is especially important in a system that implements a great deal of randomization, such as Windows Vista and Windows Server 2008.

Windows offers the ability to define a policy that restarts a failed service no more than a certain number of times or on a certain schedule. For example, an administrator could define a policy that will restart a process 10 times within 24 hours, and after that no longer allow it to restart unless an administrator physically restarts the service. It is also possible to restart a process indefinitely. But we don't want to give attackers the ability to re-try their attacks indefinitely, so we tightened up the restart policy for many highly exposed system services.

For example, in Windows Server 2008, many services, including the Network Access Protection Agent, are set to restart twice, and then no longer restart. In other words, the attacker has two shots. With all the randomization in place in Windows today, this makes the attacker's job much more difficult.

The Question of Least Privilege

You may have noticed that I have not mentioned least privilege as a defense, and I left it out on purpose. Clearly, least privilege is an important defense, but it is necessarily an imperfect wall because many products have had and will continue to have local escalation of privilege vulnerabilities. Also, least privilege does not mitigate many information disclosure vulnerabilities. Malicious code running as a normal user, rather than an administrator, can still access data accessible by the user, and that data could include sensitive data such as passwords, encryption keys, personal financial information, and e-mail. Within the SDL we think of least privilege as very important, but we also recognize that on a normal user's computer, it can be hard to enforce the security boundary and have a usable system. A good example of this is running mobile code through a Web browser. At some point, a user will probably visit a Web site that requires a Java applet, a Flash file, or perhaps some multimedia experience that will require some mobile code. Installing this code is a trusted operation, so the user must elevate to an account that can install the code. The process of elevating can lead to weaknesses in a pure least privilege environment. Of course, it is possible to utterly lock a system down in such a way that it is very difficult for a user to elevate at all; for some installations processing sensitive information, this is the right answer.

Summary

Writing code that is perfectly secure in the long term is not possible; new attack types appear almost weekly. But it is imperative that systems offer a degree of protection, even in the face of new classes of attacks and design and coding vulnerabilities. This means that software development organizations should spend a great deal of time thinking about defense in depth mechanisms, as well as focusing on "getting the code right." A simple mantra to consider is, "Your code will fail – now what?"

The problem is exacerbated by zero-day vulnerabilities, and vulnerability research moving underground to be used for criminal purposes.

If there is one lesson we can all learn, it is this: Defense in depth is just as important as following good security coding and design practices, because you will never get the product totally secure.

If there is a second lesson, it is that you must use as many defense in depth mechanisms as possible and they must be enabled by default because defense in depth is most useful in the face of an attack that takes advantage of a vulnerability that is not publicly known.

We have implemented the defenses listed above in various Microsoft products. I would urge you to take advantage of these defenses if you build on the Microsoft platform; if you use other products, understand what defense in depth mechanisms they offer and use them.◆

References

1. "Vulnerability in Exchange Server 5.5 Outlook Web Access Could Allow Cross-Site Scripting Attack." Online posting. 12 Apr. 2004 <www.microsoft.com/technet/security/bulletin/ms03-047.mspx>.
2. "Vulnerability in Exchange Server 5.5 Outlook Web Access Could Allow Cross-Site Scripting and Spoofing Attacks." Online posting. 10 Aug. 2004 <www.microsoft.com/technet/security/bulletin/ms04-026.mspx>.
3. Klein, Amit. "Divide and Conquer." HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Mar. 2004 <www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf>.
4. Howard, Michael, David LeBlanc, and John Viega. 19 Deadly Sins of Software Security. Emeryville, CA: McGraw Hill, 2005.
5. "Vulnerability in Virtual PC for Mac Could Allow Privilege Elevation." Online posting. 10 Feb. 2004 <www.microsoft.com/technet/security/bulletin/ms04-005.mspx>.
6. Howard, Michael, and Matt Thomlinson. "Windows Vista ISV Security." Microsoft Developer Network. Apr. 2007 <http://msdn2.microsoft.com/en-us/library/bb430720.aspx>.
7. Howard, Michael. "Protecting Your Code with Visual C++ Defenses." MSDN Magazine. Mar. 2008 <http://msdn2.microsoft.com/en-us/magazine/cc337897.aspx>.
8. McDonald, John. "Defeating Solaris/SPARC Non-Executable Stack Protection." Online posting. 2 Mar. 1999 <www.ouah.org/non-exec-stack-sol.html>.

About the Author

Michael Howard is a principal security program manager on the Trustworthy Computing Group's Security Engineering team at Microsoft, where he is responsible for managing secure design, programming, and testing techniques across the company. Howard is an architect of the SDL, a process for improving the security of Microsoft's software. He began his career with Microsoft in 1992 at the company's New Zealand office, working for the first two years with Windows and compilers on the Product Support Services team, and then with Microsoft Consulting Services, where he provided security infrastructure support to customers and assisted in the design of custom solutions and development of software. In 1997, he moved to the United States to work on Internet Information Services, Microsoft's next-generation Web server, before moving to his current role in 2000. Howard is a Certified Information Systems Security Professional and a frequent speaker at security-related conferences. He regularly publishes articles on security design and is the co-author of six security books, including the award-winning "Writing Secure Code," "19 Deadly Sins of Software Security," "The Security Development Lifecycle," and his most recent release, "Writing Secure Code for Windows Vista."

E-mail: mikehow@microsoft.com

Supporting Safe Content-Inspection of Web Traffic[1]

Dr. Partha Pal and Michael Atighetchi
BBN Technologies

Interception of software interaction for the purpose of introducing additional functionality or alternative behavior is a well-known software engineering technique that has been used successfully for various reasons, including security. Software wrappers, firewalls, Web proxies, and a number of middleware constructs all depend on interception to achieve their respective security, fault tolerance, interoperability, or load balancing objectives. Web proxies, as used by organizations to monitor and secure Web traffic into and out of their internal networks, provide another important example.

As more and more interactions (including personal, financial, and social) become Web-based, a number of observations can be made. First, as technology advances and public awareness of Internet security increases, an increasing portion of Web traffic is likely to be carried by Hypertext Transfer Protocol Secure (HTTPS). Second, while that will provide a level of end-to-end security, it will present a new challenge for the functions and services that rely on inspecting the content of Web traffic. Some of these services and functions will concern security, such as auditing and access control. The challenge comes from two directions: (1) the standard Web proxies of today pass the HTTPS traffic through, and (2) Web proxies are somewhat global (aggregating Web traffic from many users or applications) and agnostic to personalization to an individual user's or an application's context and requirement. We developed a *personal proxy* that is capable of handling both HTTP and HTTPS traffic, and demonstrated its use in tackling the threat of phishing attacks. This personal proxy will be a useful tool for implementing functions and services that require inspection of Web traffic content.

Introduction

The ability to intercept normal interaction between application components enabled a number of useful functions such as monitoring and auditing, adaptive failover, load balancing, and (last but not least) enforcement of security policies. Obviously, the need for many of these functions is already felt in the context of Web-based applications. The use of Web proxies by organizations to monitor and protect Web-based applications running within their networks, the use of load balancing mechanisms in server farms, and handling cross-domain exchanges are cases in point.

A number of interception-based functions require *deep inspection* of the traffic, meaning operations that need to access the content of the payload, not just the HTTP header information. Web proxies can do this job perfectly for HTTP traffic, but not for HTTPS traffic. The reason is that HTTPS is the secure version of the HTTP protocol, and HTTPS payloads are encrypted by Transport Layer Security and are not meant to be inspected or modified by interlopers like the proxy.

As important services increasingly become Web-enabled and as the task of setting up HTTPS becomes routine, we expect that increasing Web traffic will move over to HTTPS to provide a level of security that the users have come to expect (e.g., the padlock sign on the browser). This gain in one aspect of security (i.e., site authentication and defense against confidentiality and integrity attacks on the information during the transit) makes it difficult for functions that require access to the content, such as auditing and monitoring, application level rate limiting, application level adaptive caching, context-specific failover and load balancing, and so forth. In addition, as Web services become the de-facto mechanism of information exchange, proxies are likely to play a key role in handling cross-domain issues. For example, opening HTTP connection to Web sites other than the one from which the current Web page was served is usually not permitted from the browser, but the application may need to interact with services from other Web sites. Using a proxy is one solution that is often used to get around that problem. The problem gets more complicated if different services are at different security levels. If that transaction happens over HTTPS, the standard proxies will be of no use – one must use a proxy like ours that can proxy HTTPS.

"We developed a personal proxy that is capable of handling both HTTP and HTTPS traffic, and demonstrated its use in tackling the threat of phishing attacks."

The global and impersonal nature of the proxies poses another challenge. Unlike a firewall (that deals with many protocols including HTTP and many ports including those used by Web services), a Web proxy is narrowly focused on the HTTP traffic. However, like a firewall, a Web proxy covers multiple hosts, users, and applications in an aggregate form. The wide variety of Web applications and their range of importance and sensitivity – from financial transactions like banking and shopping to social interactions over Facebook, Web-based e-mail, and chat – will demand an unforeseen level of personalization or application-specificity in monitoring, auditing, access control, rate limiting, or load balancing solutions. We claim that the aggregate and one-size-fits-all nature of Web proxies will make the Proxy-based Solutions situated at the Internet Service Provider (ISP) or at corporate boundaries insufficient and less acceptable.

On one hand, the users will be less comfortable disclosing their personal preferences and requirements to the remote proxy that they do not own and control themselves. While understanding and enforcement of the policy may be a daunting task for some users, they will still demand canned policies that they can turn on. Think of setting your browser's security settings, but different settings for Facebook and your bank, and even different settings for different Facebook users in your household that you can control. Then, there will always be a group of technology-literate users questioning the adequacy of protection of personal data and the quality of enforcement offered at the remote proxy. On the other hand, because the remote proxy aggregates traffic flow from multiple users and applications, they are ill-equipped to enforce policies and preferences that are highly specialized (personalized) for individual applications or users without mutual interference.

We argue that we need a *personal Web proxy* that will do the following:

- Be situated near the user or the application it proxies (it is even possible to have dedicated proxies for each application), and is controlled by the user or the owner of the application it is proxying.
- Enforce the user's or the application's policies and personal preferences that can be easily plugged in.
- Be able to inspect HTTPS traffic without compromising the security gains contributed by the HTTPS protocol.

The envisioned personal proxy is analogous to personal firewalls: As personal firewalls bring firewall capability near to the user's host from the network edge, the personal proxy will also push proxying capability from the network edge closer to the user or the application. Furthermore, the personal proxy is a valid application level proxying mechanism that can be easily customized for the application or user at hand; it provides an easy way to introduce additional application or user-specific functionality in the HTTP/HTTPS path.

There are a number of software engineering reasons supporting the need of a separate proxy, as opposed to embedding the needed additional functions into the application components it mediates between. First, the proxy adds a separate layer of protection (another process to corrupt – a crumple zone, if you will), and provides stronger isolation guarantees (defense against memory corruption attacks) and increased flexibility. The proxy is less complex than the browser that has to support applications ranging from streaming media to Java applet, and provides a smaller attack surface. Since the proxy is a dedicated process, it can be protected using technologies that implement process protection domains, such as SELinux [1] or Cisco Security Agent [2]. Second, a personal proxy offers a good middle ground between the two extremes, dealing with the aggregate of interactions at the network edge or modifying each application. A browser plugin-based implementation will not be able to control or monitor non-browser applications that may use HTTP or HTTPS and should be subject to the same user-defined policies and preferences. To cover this situation, one either assumes (somewhat unrealistically) that all applications interacting over HTTP/HTTPS use the browser or are forced to develop similar embedded capabilities for each of those non-browser applications. Furthermore, the corporate or ISP proxy may not be able to enforce policies of individual applications and users at the network edge. It is easier to implement user- or application-specific policies and behavior into a personal proxy that runs on the user's host and, using firewall rules, mandate that the only way HTTP/HTTPS traffic gets in or out is through the proxy. Third, any mechanism that enables flexible and customizable introduction of additional behavior, constraint enforcement, and monitoring without requiring costly (and sometimes impossible) code changes in the original application is a valuable software engineering asset. The personal proxy performs this job adequately. Other than ensuring that the HTTP/HTTPS traffic flows through it, no code change is necessary for the applications that interact through it. Finally, to be general and to support all kinds of monitoring and inspection use-cases, the additional user- or application-specific policies and behavior must be inserted before traffic is encrypted with the remote site's key. To illustrate the point, note that Chinese users are able to bypass governmental scrutiny enforced at their network edge by interacting with encrypting proxies outside China. While our proposed personal proxies are controlled by the user/application it covers (as opposed to any government agency), there are use-cases (e.g., parental control, cross-domain security policy enforcement) where personal proxies provide a better solution than the browser-embedded checks or proxies at the edge.

"... a personalized proxy can be used to protect the user from divulging personal information to malicious Web sites ..."

Under Department of Homeland Security (DHS) funding, we have developed a customizable Web proxy that handles both HTTP and HTTPS protocols. For HTTPS, the proxy works by establishing two System Specification Language (SSL) connections: one between the browser and the proxy, and the other between the proxy and the remote Web site. The customization happens by configuring the proxy's chain of interceptors. The proxy can be placed near the user, on the user's computer, or at the user's home router box. We have demonstrated how such a personalized proxy can be used to protect the user from divulging personal information to malicious Web sites (i.e., defense against phishing attacks). We have started investigating other uses of the proxy, such as auditing inter-agent communication in a semantic Web application so that the recorded interactions can be used by machine-learning algorithms that aim to learn and improve how the agents achieve their tasks. In this article, we briefly describe the architecture and operation of this personal proxy; a detailed description and the anti-phishing application appears in [3].

Architecture of the Personal Proxy

Figure 1 illustrates the design of the personal proxy, which consists of four main modules that are implemented on top of Jetty, a popular open-source Web server written in Java [4]. The *plugin framework* provides a means for integration of custom reactive and proactive behavior. In the first application of this proxy, all anti-phishing checks were implemented as a set of plugins for this module. A plugin can be one of the following three types, depending on its role in the overall control flow and threading logic:

- **Data plugins.** Each data plugin is invoked on every request and associated response. A data plugin is used for handling the header and payload data based on a specified security policy. For example, a proxy could be configured to record all or selected parts of Web traffic as part of a parental control policy. Recordings can be persisted securely on the disk.
- **Checks.** These plugins are organized in a chain, and intercepted requests flow through these checks like a pipeline. An individual check exits with either a *break* or a *continue*. A *continue* indicates that the request goes to the next stage, possibly with some additional metadata tagged to it. *Breaks* can be of two kinds: A negative break indicates that the request is to be blocked, while a positive break indicates that the request is to be accepted. In either case, a break implies that the rest of the pipeline stages are not executed.

This semantics of checks is amenable to modular implementation and integration of security policies.

- **Probes.** In contrast to checks and data plugins, which only execute reactively when triggered by requests or responses, probes allow us to embed proactive behavior into the proxy. Probes contain dedicated threads that trigger monitoring functions at regular configurable intervals. The probes can be configured to visit specified URLs and scheduled intervals to collect data that is relevant for the security policy context. For example, in the case of defending against phishing attacks, the probes were used to check for changes in an Internet Protocol address or security credential of the banks or financial sites registered by the user.

The lower part of Figure 1 displays the remaining three modules. The modules act as access paths into the proxy. The *HTTP Proxy* listens on a configurable network port (e.g., 8080) for incoming HTTP requests, and dispatches the requests to a main handler (InterceptHandler), which in turn makes strategic use of the plugins. This flow is similar in the case of the *HTTPS Proxy*, except that it listens on a different network port (8443) and uses a custom extension of the InterceptHandler (called SslProxyHandler) that intercepts HTTPS connect requests and facilitates subsequent interception of all HTTPS requests in that session. The third access path, HTTPS Requests, is for management of the proxy through an administration console. Management functions include changing the order of plugins and their respective importance weights as well as customization of user-specific data. The administrative interface is optional for out-of-the-box deployment, where the proxy is preconfigured and preloaded with appropriate plugins that enforce the desired policy. We do not anticipate that the internal details are important for most of the users (beyond pointing their applications or browsers to the proxy). The users who write and package custom policies for different users and applications will need to know the details of plugins. A better policy interface, supporting a generation of plugins (which can be added to the proxy by editing a configuration file) from higher-level policy specification, and a better way to inspect the policies encoded in existing plugins, is part of our future work. Once this policy interface is in place, these users will also be shielded from the internal details and complexities of the plugin architecture. If the internal details change because of evolution of the Jetty code base/Web services specification, only the policy interface implementation will need to change.

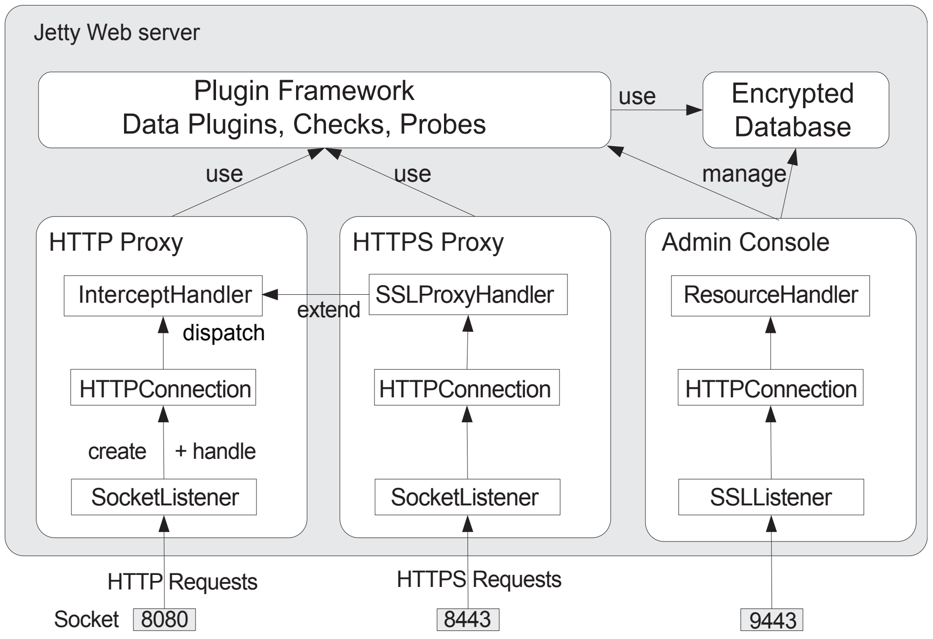


Figure 1: *Functional Architecture of the Personal Proxy*

Placement Options

The standard deployment of the proxy is on the end user's computer. Although this puts a small load on the Central Processing Unit (CPU), memory, and disk resources on the end system, it has the benefit of putting the proxy under direct control of the end-user. Our understanding is that end-users feel uncomfortable with disclosing personal and sensitive information (preferences, policies) to external parties, but are more amenable to providing this information to local components as long as it doesn't leave their machine. Since many end-users own either a wireless or DSL router and since these devices already ship with Web server capabilities, we investigated deploying the proxy on a Linksys WRT54G wireless router running OpenWrt [5]. Another option is to run the proxy on a home router, which has the benefits of increased security through stronger isolation from a potentially virus-infected desktop, and a new value-add for router manufacturers. On the downside, the very limited CPU and memory resources of the home routers, especially wireless routers, significantly lowers the performance of the proxy.

Insertion Into HTTP(S) Flow

Insertion of the proxy into the non-encrypted HTTP client-server path is straightforward and involves changing the client application's proxy settings (e.g., HTTP Web browser). To prevent an attacker from replacing the proxy setting to a proxy of his own, and to ensure that any application using HTTP/HTTPS is subject to the security policy enforced by the personal proxy, firewall rules should be set to only allow outgoing Web traffic through the personal proxy. For intercepting encrypted requests from client application that uses HTTPS, the client application's (such as the browser's) proxy settings are changed accordingly to redirect requests to personal proxy's HTTPS port. However, describing how appropriate security associations are established is slightly more involved (see Figure 2, next page).

In a regular use case without any HTTPS proxy, SSL relies on a Public Key Infrastructure for connection establishment [6]. Following a general description of the SSL protocol, the client issues a connection request to the server, which the server acknowledges with a response containing a certificate signed by a certification authority (CA). The client then continues to perform a set of checks on the server certificate, the main one of which is to verify that the CA's signature is valid. In most cases, SSL transactions essentially establish a unidirectional trust relationship between the browser and the target Web server via a commonly trusted CA.

With the proxy in the mix, the protocol becomes a little more complex. The proxy takes on the role of a server when communicating with the browser and the role of a browser when communicating with the target Web server. This requires the proxy to dynamically generate X509 certificates for each Domain Name

Coming Events

October 14-16
Software Assurance Forum
Gaithersburg, MD
https://buildsecurityin.us-cert.gov/daisy/bsi/events/930-BSI.html

October 19-23
International Conference on Object-Oriented Programming Systems, Languages, and Applications
Nashville, TN
www.oopsla.org/oopsla2008

October 20-21
National Defense Industrial Association Technical Information Division Conference
Huntsville, AL
www.ndia.org/meetings/9010

October 26-30
SIGAda with SAMATE
Portland, OR
www.sigada.org/conf/sigada2008

November 10-14
Agile Development Practices 2008
Orlando, FL
www.sqe.com/agiledevpractices

November 11-14
19th IEEE International Symposium on Software Reliability Engineering
Seattle, WA
www.csc2.ncsu.edu/conferences/issre/2008

December 8-12
Annual Computer Security Applications Conference
Anaheim, CA
www.acsac.org

April 20-23, 2009
STC Systems & Software Technology Conference
21st Annual Systems and Software Technology Conference
Salt Lake City, UT
www.sstc-online.org

Coming Events: **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**

System name it is proxying[2] certified by its own CA[3] (called PB CA in Figure 2). During installation, the Web browser's (and any other application's using HTTPS) settings are configured to trust signatures from the PB CA. As a result, the overall trust relationship between browser and target Web server can now be decomposed into two daisy-chained relationships, one between the browser to the personal proxy, and a second between the personal proxy and the target Web server.

Does the proxy introduce additional security vulnerabilities by breaking the end-to-end encryption between browser and Web server? The answer to this question depends on the relative trustworthiness of the proxy compared to the browser and target Web server and where it is deployed. Consider the case where the user does not use a personal proxy, but thinks that his desktop and the servers he uses are more secure than the ISP server through which he uses the Internet. The ISP server may co-host other applications, and if it does not have the latest security patches installed, such a setup would significantly lower the overall security of Web transactions flowing through it. On the other hand, if the personal proxy is co-located with the Web browser on the same desktop, we would expect it would be more difficult for attackers to subvert or corrupt the Java-based stand-alone proxy process (which only listens on localhost) compared to a C++ Web browser running Javascript. In both cases, data is never sent unencrypted over the network, so the guarantees provided by SSL across host boundaries are not affected.

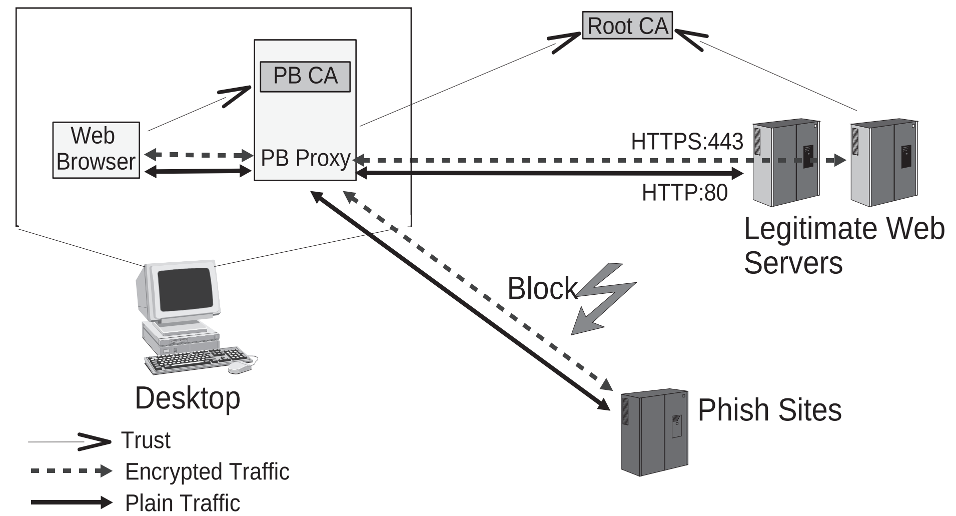
Performance Overhead

Introduction of a clearly noticeable delay presents increased resistance to adoption of the new technology. To minimize performance impact, we implemented the proxy on top of the high performance Jetty Web server and implemented various optimizations in the SSL proxy architecture to keep request latencies (i.e., elapsed time between a request and its response) within user acceptable levels. In this section, we use *overhead* to mean the increase in request latency due to interception of HTTPS traffic by the proxy.

We measured the overhead in a lab setting by visiting HTTPS sites without the proxy and with the proxy configured with a number of anti-phishing checks. The mean time to load the visited pages through the proxy was twice the mean time to load the same pages without the proxy (excluding any user interaction like typing a password for both cases). However, the variance of load time was comparable to the mean (not surprising because we were visiting sites on the Internet), and even an overhead of roughly 100 percent was not distinguishable from the noise (as noted by external field testers, the delay introduced by the proxy does not noticeably impact the user Web surfing experience). Much of this overhead can be attributed to crypto operations and session multiplexing performed in Java. We expect the plumbing overhead to stay independent of the policy checks enforced by the proxy.

We also compared the round-trip latencies between an auditing configuration (when the proxy is simply recording) and a policy enforcing configuration (loaded with anti-phishing checks). We found that the two distributions are not significantly different as their inter-quartile ranges overlap to a large extent (from 200 to 1,500 milliseconds [ms]) and both distributions have a large number of outliers (some even greater than

Figure 2: *Personal Proxy as a Trusted Middleman*



50,000 ms). We suspect that available network bandwidth to the external Web sites together with available CPU resources of those sites have the biggest impact on round trip latencies, which is why the distributions looked similar.

Related Work

Various HTTP and HTTPS proxy implementations exist for debugging purposes (Burp proxy [7], Charles proxy [8]) and Web filtering (WebCleaner [9], Privoxy [10]). There are also a number of commercial network layer tools (e.g., eSafe's Web Threat Analyzer [11], McAfee IntruShield [12]) that can inspect Web traffic, including HTTPS that work at the enterprise layer. In many cases, these are geared for regulatory and auditing compliance, the DHS-funded research focused on transparent inspection of SSL traffic exclusively for regulatory purpose. However, we were unable to find a proxy that could be used as a general purpose middleware construct for customized user and application-specific policies.

Conclusion

We have been developing advanced middleware technologies that enable adaptive behavior, quality of service (QoS) management and QoS-based adaptive behavior in distributed systems over the past several years [13]. In doing so, we have developed middleware constructs for handling different styles of distributed interaction (e.g., distributed objects, publish-subscribe, group communication) over a number of protocols (e.g., socket-based, Common Object Request Broker Architecture or Remote Method Invocation). The present work involving HTTP and HTTPS interception complements that line of successful work, and enables us to introduce advanced middleware capability to distributed systems that use these protocols. The concept of a personal proxy has the potential to fill an important and emerging gap in the current Web-based systems architecture.

However, as noted earlier, the personal proxy is still in its early stages – we only have a prototype implementation that is demonstrated with anti-phishing checks, and have just begun exploring its use in other contexts.

A number of software engineering and usability issues also need additional work, including an easy way to inspect enforced policies and the ability to define policies at a higher level of abstraction that can be automatically translated into executable code that can be integrated into the plug-in framework. These are the next steps we hope to tackle.◆

References

1. Loscocco, P., and S. Smalley. Integrating Flexible Support for Security Policies Into the Linux Operating System. Proc. of 2001 USENIX Annual Technical Conf. USENIX Association, Berkeley, CA: 2001.
2. Cisco. "Cisco Security Agent-Enterprise Solution for Protection Against Spyware and Adware." Cisco White Paper <www.cisco.com/en/US/prod/collateral/vpndevc/ps5707/ps5057/prod_white_paper0900aecd8020f438.html>.
3. Zodgekar, Sameer A. Identity Theft: A High-Tech Menace. ICFAI University Press. Apr. 2008.
4. Mortbay.com. The Jetty Java Web Server Vers. 5.1. 2007 <www.mortbay.org/jetty-6/>.
5. Openwrt.org. The Linux Distribution for Wireless Freedom <http://openwrt.org>.
6. Wagner, D., and B. Schneier. Analysis of the SSL 3.0 Protocol. Proc. of the Second USENIX Workshop on Electronic Commerce. Oakland, CA: 1996.
7. Portswigger.net. The Burp Proxy Tool Vers. 1.1. 2008 <www.portswigger.net/proxy>.
8. Charles Web Debugging Proxy. About Charles. <www.charlesproxy.com>.
9. Kuhnast, Charly. "Junk Zapper." Linux Magazine June 2004 <www.linux-magazine.com/issue/43/Charly_Column.pdf>.
10. The Privoxy Team. Privoxy 3.0.8 User Manual. 2008 <www.privoxy.org/user-manual/index.html>.
11. Aladdin.com. "The eSafe Web Threat Analyzer Audit." 2008 <www.aladdin.com/esafe/solutions/wta>.
12. McAfee.com. "McAfee Network Security Platform Data Sheet." 2007 <www.mcafee.com/us/local_content/datasheets/ds_network_security_platform.pdf>.
13. BBN Technologies. The QuO Group at BBN. "Distributed Systems Technology Group Papers." <www.dist-systems.bbn.com/papers/>.

Notes

1. This work was supported by the DHS Advanced Research Projects Agency under contract number NBCHCO50096.
2. To increase generation performance, key pairs can be reused across certificates.
3. Alternatively, the PB CA can be signed by a common root CA.

About the Authors

Partha Pal, Ph.D., is a division scientist at BBN Technologies' National Intelligence Research and Application business unit. His research interests include adaptive and survivable distributed systems and applications, and technologies that enable adaptive behavior and survivability. Pal has published more than 35 technical papers in peer-reviewed journals and conferences and is a senior member of the Institute of Electrical and Electronic Engineers (IEEE) and a member of the Association for Computing Machinery. He received his master's and doctorate degrees from Rutgers University, New Brunswick, NJ.

BBN Technologies
10 Moulton ST
Cambridge, MA 02138
Phone: (617) 873-2056
Fax: (617) 873-4328
E-mail: ppal@bbn.com

Michael Atighetchi is a scientist at BBN Technologies' National Intelligence Research and Application business unit. His research interests include security and survivability, intelligent agents, and middleware technologies. Atighetchi has published more than 20 technical papers in peer-reviewed journals and conferences, and is a member of the IEEE. He holds a master's degree in computer science from University of Massachusetts at Amherst, and a master's degree in information technology from the University of Stuttgart, Germany.

BBN Technologies
10 Moulton ST
Cambridge, MA 02138
Phone: (617) 873-1679
Fax: (617) 873-4328
E-mail: matighet@bbn.com

Enhancing the Development Life Cycle To Produce Secure Software

Karen Mercedes Goertzel
Booz Allen Hamilton

Over the past decades, efforts to enhance software development life cycle (SDLC) practices have been shown to improve software quality, reliability, and fault-tolerance. More recently, similar strategies to improve the security of software in organizations such as Microsoft, Oracle, and Motorola have resulted in software products with less vulnerabilities and greater dependability, trustworthiness, and resilience. In its mission to improve the security of software used in America's critical infrastructure and information systems, the Department of Homeland Security's (DHS) Software Assurance Program has sponsored the creation of the book Enhancing the Development Life Cycle to Produce Secure Software*, a source of practical information intended to help developers, integrators, and testers identify and systematically apply security and assurance principles, methodologies, and techniques to current SDLC practices, and thereby increase the security of the software that results. Unlike the numerous other books on secure software development,* Enhancing the Development Life Cycle *does not espouse any specific methodology, process model, or development philosophy. Instead it explains the essentials of what makes software secure, and takes an unbiased look at the numerous security principles and secure development methodologies, practices, techniques, and tools that developers are finding effective for developing secure software – information that readers can leverage in defining their own SDLC security-enhancement strategies.*

In its report to President George W. Bush entitled "Cyber Security: A Crisis of Prioritization" (February 2005), the President's Information Technology Advisory Committee summed up the problem of non-secure software:

> Network connectivity provides "door-to-door" transportation for attackers, but vulnerabilities in the software residing in computers substantially compound the cyber security problem…. Software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit. Today, as with cancer, vulnerable software can be invaded and modified to cause damage to previously healthy software, and infected software can replicate itself and be carried across networks to cause damage in other systems.
>
> Like cancer, these damaging processes may be invisible to the lay person even though experts recognize that their threat is growing. And as in cancer, both preventive actions and research are critical, the former to minimize damage today and the latter to establish a foundation of knowledge and capabilities that will assist the cyber security professionals of tomorrow to reduce risk and minimize damage for the long term. Vulnerabilities in software that are introduced by mistake or poor practices are a serious problem today. In the future, the nation may face an even more challenging problem as adversaries – both foreign and domestic – become increasingly sophisticated in their ability to insert malicious code into critical software.

Software is considered "secure" when it exhibits three interrelated properties:

1. **Dependability.** The software executes correctly and predictably, even when confronted with malicious or anomalous inputs or stimuli.
2. **Trustworthiness.** The software itself contains no malicious logic or any flaws or anomalies that could be exploited or targeted as vulnerabilities by attackers.
3. **Resilience.** When the software is able to resist most attempted attacks, tolerate the majority of those it cannot resist, and recover with minimal damage from the very few attacks that succeed (i.e., those the software could neither resist nor tolerate).

A number of factors influence how likely software is to consistently exhibit these properties under all conditions. These include:

- The programming language(s), libraries, and development tools used to design, implement, and test the software, and how they were used.
- How the software's re-used, commercial off-the-shelf, and open source components were evaluated, selected, and integrated.
- How the software's executables were distributed, deployed, configured, and sustained.
- The security protections and services provided to the software by its execution environment.
- *The practices used to develop the software, and the principles that governed those practices.*

Experience over the past few decades has shown that enhancing SDLC practices with the objective of improving software quality, reliability, and fault-tolerance does, in fact, result in software that is higher in quality, or more reliable, or more tolerant of faults. More recently, the same SDLC enhancement approach has been applied to improve the *security* of software. By adjusting and, in some cases expanding, SDLC activities to ensure that they consistently adhere to secure specification, design, coding, integration, testing, deployment, and sustainment principles, organizations such as Microsoft, Oracle, Motorola, Praxis High Integrity Systems, and a growing number of others, have been able to report that soon after doing so, they were finding vulnerabilities and weaknesses much earlier in the software's life cycle. In turn, those organizations were able to eradicate problems at a much lower cost than ever before. Moreover, the organizations that institutionalized repeated use of *security-enhanced* SDLC practices found that, over time, the enhanced practices became second-nature to their developers, and fewer and fewer vulnerabilities and weaknesses appeared in their software in the first place. Not only that, but they also noted that their software's dependability, trustworthiness, and resilience also improved.

Enhancing the Development Life Cycle to Produce Secure Software

This year, the DHS sponsored the revision of its 2006 document, "Security in the Software Life Cycle: Making Development Processes – and Software Produced by Them – More Secure[1]." The new document, retitled "Enhancing the Software Life Cycle to Produce Secure Software," transforms what was essentially a compendium of software security assurance concepts and overviews of methodologies, process models, sound practices (also known as *best practices*), and supporting technologies that, when used, had been reported by their advocates to produce software that is more secure than software built by more traditional methods and tools.

By contrast, the authors of "Enhancing the Software Life Cycle" have transformed their previous survey of the software assurance domain into a source of practical information to enable developers, integrators, and testers to identify and systematically apply security and assurance principles, methodologies, and techniques to enhance their current SDLC practices. The revision's focus has been narrowed and its concept discussions streamlined, while its pragmatic technical content has been expanded and deepened and augmented with extensive lists of references to information (online and in print) on how to implement the various techniques and methodologies described in the document. The new version also reflects pertinent technological, methodological, and philosophical advances that have occurred in the software and software assurance domain since the release of "Security in the Software Life Cycle" more than two years ago.

Several other developments made this revision possible. The Department of Defense (DoD) Technical Information Center (DTIC) sponsored the public release of "Software Security Assurance: A State-of-the-Art Report," which captured and updated much of the survey-type information in "Security in the Software Life Cycle." In addition, DHS produced its drafts of "Software Assurance in Acquisition[2]," and "Practical Measurement Guidance for Software Assurance and Information Security[3]." The DoD drafted "Engineering for System Assurance"[4], the DTIC-sponsored "Project Management for Software Assurance: A State-of-the-Art Report,"[5] and Addison-Wesley published "Software Security Engineering: A Guide for Project Managers" as part of its Software Security Series[6]. Collectively, these publications address many concerns of the secondary intended audience for "Security in the Software Life Cycle," including acquisition managers, project managers, system engineers, and information security practitioners, enabling "Enhancing the Software Development Life Cycle" to focus on its primary audience of developers, integrators, and testers.

Unlike the steadily increasing number of other books on secure software development, secure programming, application security, and software security testing, "Enhancing the Development Life Cycle" strives to remain *methodology/process-agnostic*. Its intent is to explain the essentials and characterize the advantages of a number of security principles, and secure development methodologies, practices, and techniques that have proven effective in the security-enhancement of SDLC activities.

"Enhancing the Development Life Cycle" is currently undergoing a final review by the DHS/DoD co-sponsored Software Assurance Working Groups. Based on comments from those reviewers, the final draft will be produced for public comment late this summer. It will be available for download from the Build Security In Web site at <https://buildsecurityin.us-cert.gov/daisy/bsi/dhs.html>.◆

Notes

1. Goertzel, Karen Mercedes, and Joe Jarzombek. "Security in the Software Life Cycle." CrossTalk. Sept. 2006. Accessed 30 May 2008 at <www.stsc.hill.af.mil/crosstalk/2006/09/0609Jarzombek Goertzel.html>.
2. Accessed 30 May 2008 at <https://buildsecurityin.us-cert.gov/daisy/bsi/908.html?branch=1&language=1>.
3. Accessed 10 June 2008 at <https://buildsecurityin.us-cert.gov/swa/downloads/SwA_Measurement.pdf>.
4. Accessed 30 May 2008 at <www.acq.osd.mil/sse/ssa/docs/SA+Guidebook+v905-22Apr08.pdf>.
5. Abstract accessed 30 May 2008 at <https://www.thedacs.com/techs/abstracts/abstract.php?dan=347617>.
6. Mead, Nancy R., et.al. Software Security Engineering: A Guide for Project Managers. Upper Saddle River, NJ: Addison-Wesley, 2008.

About the Author

Karen Mercedes Goertzel, Certified Information Systems Security Professional, is a subject-matter expert in software assurance, cyber security, and information assurance. She supports the DHS' software assurance program, not least as lead author/editor of "Security in the Software Life Cycle." Goertzel coordinated the Information Assurance Technology Analysis Center (IATAC)/ Data and Analysis Center for Software "Software Security Assurance: A State-of-the-Art Report," as well as IATAC's "The Insider Threat to Information Systems: A State-of-the-Art-Report." She was a contributing author to the National Institute of Standards and Technology Special Publication 800-95, "Guide to Secure Web Services," editor/contributing author of the National Security Agency's "Guidance for Addressing Malicious Code Risk," and has been published previously in CrossTalk. As lead technologist for the Defense Information System Agency's Application Security Project from 2002-2004, she was the leading contributor to the DoD Application Security Developer's Guides upon which the Defense Information System Agency's Application Security and Development Security Technical Implementation Guide is (in large part) based. Goertzel has presented at numerous conferences and workshops, including the American Institute of Engineers' 2008 Military Open Source Software Conference and the DoD's 2008 System and Software Technologies Conference. Before joining Booz Allen Hamilton, she was a specialist in the specification and architecture of high assurance cross-domain information sharing solutions for defense and civil establishments in the U.S., Canada, Australia, and NATO.

Booz Allen Hamilton
8283 Greensboro DR
H5061
McLean, VA 22102
Phone: (703) 902-6981
Fax: (703) 902-3537
E-mail: goertzel_karen@bah.com

Hazardous Software Development

Corey P. Cunha
Savard Engineering

Developing safety-critical software is often an extremely complicated process, and if managed incorrectly could have the tendency to cause more harm than good. In order to deal with the challenge of writing safety-critical software, certain considerations must be followed. Different case studies will be used in this article to illustrate points about the ethics standards, hazard identification challenges, and aftermath management techniques needed to effectively manage the development and deployment of safety-critical software.

Safety-critical systems are important in everyday life and are used to manage difficult tasks that may otherwise be impossible to do. Many industries use systems that pose potential hazards to the general public, such as systems designed for the aviation and social engineering industries. When systems are developed for use in hazardous situations, development strategies and ethics standards must change to fit the needs of the project in order to ensure the safety of employees and the general public. As the demand for these systems continues to grow, standards and strategies developed to keep safety-critical software safe will continue to evolve.

Case Studies

To illustrate certain points, incidents involving failures in Union Carbide plants, the Patriot Missile Defense System, Iran Air Flight 655, and Therac-25 will be analyzed.

Union Carbide

Around midnight on December 2, 1984, a large amount of water was introduced into a Methyl Isocyanine (MIC) storage tank at Union Carbide's Bhopal, India pesticide plant, producing one of the worst industrial accidents in history. Once mixed with water, the MIC produced a vapor that escaped through the ventilation system into the atmosphere. Quickly afterward, the MIC began to decompose into many different components, including cyanide gas, and "... the immediate death toll from the cyanide release was in excess of 2,000 people" [1]. In the years to follow, the total death and injury counts ranged in the hundreds of thousands.

Though explanations vary, most attribute the disaster to a combination of human error and faulty safety systems. The system also lacked software safeguards that would prevent water mixture if certain hardware safeguards were not in place.

Patriot Missile Defense System

In 1991, a software design error in the Patriot Missile Defense System caused the deaths of 28 U.S. soldiers when a missile was fired at the wrong destination. After investigation, it was determined the error was due to differences in the clocks of the internal missile system and the global system, resulting in a miss when intercepting an incoming missile.

Iran Air Flight 655

Iran Air Flight 655, a civilian air flight traveling on an approved flight path, was shot down by a U.S. guided missile cruiser on July 3, 1988. A U.S. Navy investigation discovered the plane was shot down because of both a bug in the Advanced Electronic Guidance and Instrumentation System (AEGIS) aboard the craft along with bad decision-making skills by the commanding officer.

At no time did the AEGIS system fail completely; the order to fire was given by the commanding officer because of operator error and lack of judgment. Once the aircraft was identified by AEGIS, the system then proceeded to ask the aircraft for its friend or foe (FOF) status. The commercial aircraft then responded with the code *commair*, identifying itself as a commercial aircraft. After trying to confirm the FOF status with repeated radio communication, warning signals were given to have the aircraft change course.

Without any response from the aircraft, AEGIS was instructed to read the FOF signal from the plane again. This posed a problem as AEGIS could not read the FOF signal from the same plane twice without rebooting, causing the system to read the signal of a fighter jet stationed at a nearby air base. With a new signal indicating a military aircraft, the commander issued the order to fire.

Therac-25

Between June 1985 and January 1987 Therac-25, a Medical Linear Accelerator designed by Atomic Energy of Canada Limited (AECL), malfunctioned multiple times. These malfunctions resulted in the deaths of three patients, with many more being injured.

Unlike previous inventions by AECL, the Therac-25 did not use proven hardware with interlocks to ensure safety. Instead, Therac-25 relied completely on software modified from older systems in order to reduce costs [2]. Because of this design choice, there was no safety protocol when the software failed.

Based on user feedback, AECL modified the software before the initial release to include a quick re-entry method that allowed the operator to skip re-entry of certain treatment information [2]. This step was initially used to ensure the patient was getting the correct dosage prescribed. However, the change was not thoroughly tested, and as a result a software bug was never found.

If used extremely quickly, the system would generate an error and subsequently notify the operator of a cryptic error message. After resetting the device to clear the error, the machine would not indicate any dosage was applied, but the patient would receive a hazardous overdose of electrons.

Identifying Hazards

Hazard analysis, though important in every software engineering discipline, has an extremely pronounced role when dealing with safety-critical systems. There are several methods to perform hazard analysis, but before analyzing the hazards they must be identified.

As stated in [3], "There does not appear to be any easy way to identify hazards within a given system." Because of this fact, many identification methods are often used in order to find as many hazards as possible in a given system. The primary techniques used in this process are the Delphi Technique and joint application design (JAD).

Delphi Technique

The Delphi Technique, developed by the

Rand Corporation in the early 1980s, is an identification technique that aims at gathering different ideas from different geographical locations.

The advantages of this technique are numerous. The technique requires anonymity, reducing the chance one voice will be heard over the opinions of others. This technique also eliminates the need for participants to congregate, removing the stress of arguments between participants and the struggle of a group reaching a consensus.

Even with its advantages, the Delphi Technique can be slow to administer. The technique uses questionnaires, polls, and other traditional information-gathering techniques. Because of this limitation, the Delphi Technique calls for multiple iterations, slowing down the process. The process could also be slowed by the amount of workload on the moderator.

JAD

JAD is a hazard brainstorming system pioneered by IBM. Generally, JAD meetings are used in order for a group of people to reach an agreement on what types of hazards could occur and how those hazards could affect the system or a person's quality of life [3].

JAD development, though fast and thorough, was originally designed for other uses then adapted to be used in hazard identification. Because of this, JAD does not seem to identify hazards as well as other development styles. JAD also forces a consensus on the people involved in the process, which is often difficult to reach.

Analyzing Hazards

Hazard analysis is the process of examining all identified hazards, with the goal of locating the points of failure which cause the hazards. There are two different types of analysis: inductive, and deductive techniques. Inductive techniques attempt to start with answers and generate the questions during the analysis process, while deductive techniques look for answers to questions that are pre-defined. This article will focus only on inductive techniques, though, including event tree analysis, failure modes and effects analysis, hazard and operability analysis, and fault tree analysis.

Fault Tree Analysis

Fault tree analysis focuses on one hazard and works to find any events that could trigger the hazard to occur. The hazard in question is placed at the top of the tree with each subsequent event that could trigger the main hazard underneath it.

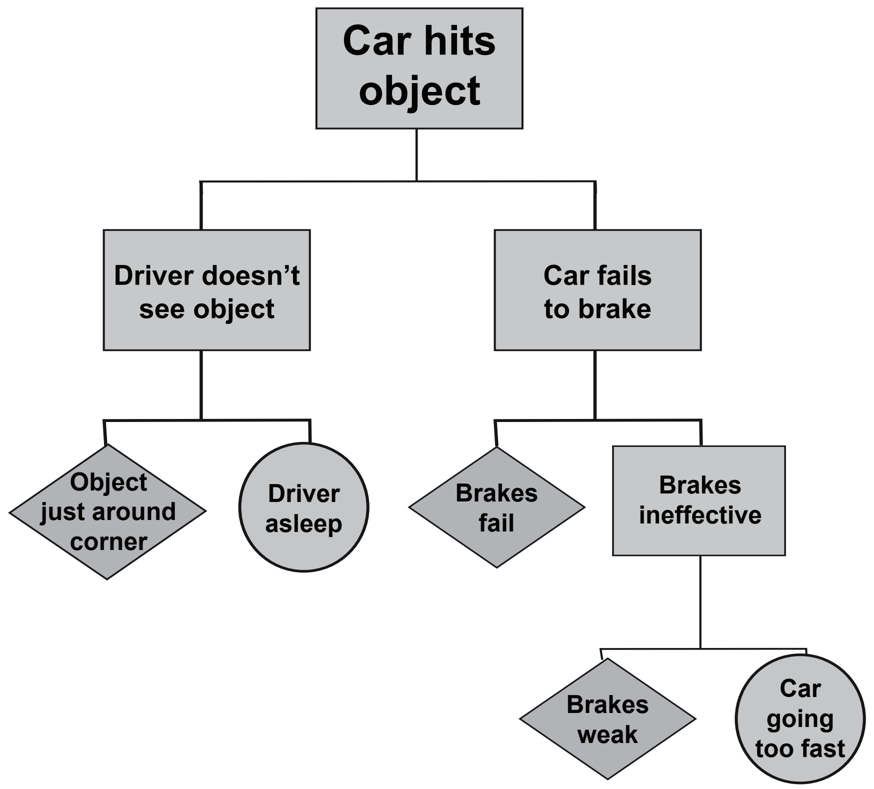


Figure 1: *A Sample Fault Tree Analysis Document*

Figure 1 shows an example fault tree diagram of a car where a vehicle collision is the final outcome. The diagram identifies many points of failure, such as the brakes and the driver's ability. While driver ability cannot be controlled in design, the brake design can now be prioritized to ensure safety.

A fault tree diagram can serve many purposes, the most important being the ability to determine different factors to a system failure. Fault tree diagrams can also be used during all the phases in a software life cycle, reducing the chances of missing a hazard after the planning stages.

The main downside to fault tree analysis is the high cost of producing documentation. Therefore, it is rarely used outside of high-risk situations [3].

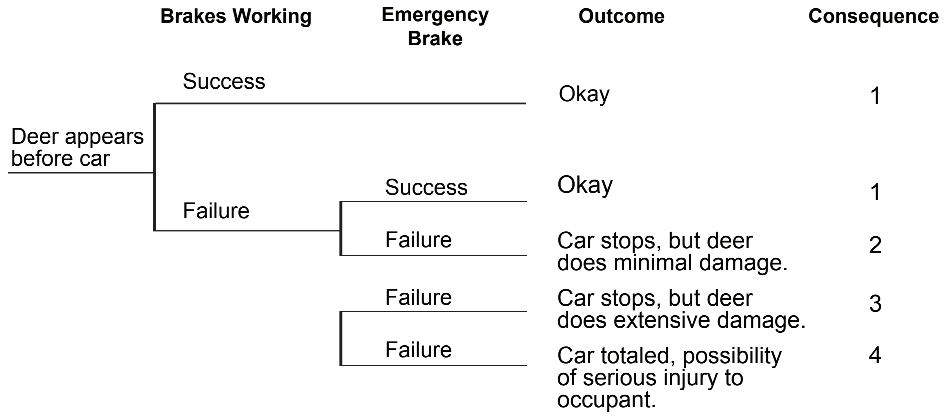
Event Tree Analysis

Event tree analysis has many similarities to fault tree analysis, but instead focuses on finding the outcomes of a system failure. Figure 2 shows an example of an event tree diagram for a car brake system. This specific example looks at a braking hazard, with the main brake and emergency brakes being the points of failure. Branching at each failure point, system developers can see the outcome of almost every failure situation.

Hazard and Operability Analysis

Hazard and operability analysis (HOA), or operating hazard analysis, is a hazard

Figure 2: *A Sample Event Tree Analysis Document*



Brakes Working
Emergency Brake
Outcome
Consequence
Deer appears before car
Success
Okay
1
Failure
Success
Okay
1
Failure
Car stops, but deer does minimal damage.
2
Failure
Car stops, but deer does extensive damage.
3
Failure
Car totaled, possibility of serious injury to occupant.
4

analysis technique that requires attention during all stages of the software engineering development cycle. By using HOA, a project manager can ensure the maximum number of hazards are identified and corrected before product release [3].

HOA uses two different steps in order to analyze hazards. First, each hazard is either classified into being a human hazard or an environmental hazard. After grouping the hazards, they are analyzed further to generate a list of problems that can result from each specific hazard. Each point in the process is then checked to see if it is a safety-critical concern. If deemed safety-critical, the hazard is analyzed to determine how to minimize the possibility of its occurrence.

This type of hazard analysis could have been used for the system at Union Carbide. By searching for potential hazards throughout every step of the design process, the development team may have found more points of failure in the system and designed safeguards around these failure points.

Failure Modes and Effects Analysis

Failure modes and effects analysis was first developed in 1984 by the Department of Defense as a measure to prevent hazardous failures of critical systems. This method uses a table structure to take every component of the system and evaluate different failure modes for that component. This type of analysis also is used to find [3]:

- The effect of component failure.
- The cause of component failure.
- The occurrence chances.
- The severity of the problem.
- The probability of detecting the problem before the hazardous situation occurs.

Figure 3 shows an example of a failure modes and effects document based around a vehicle tie bar bracket. By analyzing all possible failure modes, designers of the tie bar bracket can see that invalid specifications of certain parts could cause the bracket to fail.

Examples

The blame for Bhopal was primarily placed on the shoulders of the developers of the system, not on the operators themselves [1]. Several ways are listed that the development team could have been better prepared for when working on a safety-critical system. Some points described in [4] include:

- Setting learning objectives for the developers, such as:
  - Assigning a design manager to ensure safety precautions are taken.
  - Creating a safety program to oversee the project.
  - Conducting random audits of software.

"When a safety-critical system does fail, it then becomes the responsibility of the company to follow up and take charge of the problem."

- Making sure employees are receiving education and not *shrugging off* their responsibilities.
- Learning from past mistakes of system failures.
- Including safety-critical engineering topics into software engineering courses.

A couple reasons why safety is often given second-place status to other factors in design is provided in [5]. Because safety issues often are ignored or unknown, these issues generate a substantial amount of risk to product success and personal safety. These are risks outlined in [3] which must be taken into consideration when doing both hazard identification and analysis. Some other risks include the following:

- Developers not conforming to the safety engineering processes due to time restraints.
- Developer lack of safety-oriented training.
- Lack of automated safety shutdown procedures in an emergency situation.

Conclusion

Had the design team at Bhopal used one of these development strategies, they could have been more prepared for the conditions which caused the 1984 disaster. If safety education and safety-minded development were taken into account earlier in the design process, the system may have prevented the operator from mixing the solution unless the certain safety conditions were met.

Hazard identification and analysis is extremely important in any safety-critical system, allowing for potential hazards to be found and analyzed before they cause problems in a final product.

These steps also go hand-in-hand with verification and validation (V&V) steps defined by the Institute of Electrical and Electronics Engineers (IEEE) standards document 1012. This document defines guidelines on how to first identify and manage hazards and risks. V&V steps ensure that problems are documented and then followed through, resulting in safer systems.

Codes of Ethics

With the amount of safety-critical systems currently in use, ethics play a larger role in software engineering today than they ever have in the past. The Computer Society, in cooperation from the IEEE, put together a set of ethics codes which all software engineers and designers should follow in order to assure software quality and safety. The standards relating to hazardous development discussed in [4] include, but are not limited to:

- Assuring that software engineers working on a project take responsibility for the code they write or decisions they make. For example, AECL never revealed the developer of the Therac-25 system; therefore, there was never a person to question about the software issues. This is often hard to enforce, as the original engineer may not be part of the company anymore.
- Approving software only if it meets specifications of safety and performance, passes all tests required, and

Figure 3: *A Sample Failure Modes and Effects Analysis Document for a Tie Bar Bracket*

Failure Mode	Effect of Failure	Cause of Failure	Occurrence	Severity	Probability of Detection
Bracket fractures	Stabilizing function of tie bar removed. All engine motion transferred to mountings.	Inadequate specification of hole-to-edge distance	1	7	10
Bracket corrodes	As above	Inadequate specification for preparation of bracket	1	5	10
Fixing bolts loosen	As above	Bolt torque inadequately specified	5	5	8
As above	As above	Bolt material or thread type inadequate	1	5	10

does not hinder the life or safety of a person.

- Ensuring that any end-user documentation for the software is fully prepared. This was also a problem in the Therac-25 system, as AECL did not include definitions of their error messages inside of the operator's manual. Because of this, many users of the system just bypassed errors the system displayed.
- Disclosing any risks or hazards known about the system prior to the installation of the product.
- Cooperating with any concerns directed at the company regarding safety. This includes fully investigating any safety claims. In the case of AEGIS, the company quickly changed their systems over to the Linux operating system in order to fix rebooting problems. Therac-25 did, in some respects, cooperate with the customers but refused to take any extra actions until it was proved that their product was causing problems.

Besides the examples mentioned, Union Carbide possibly violated ethical standards by not adding safety functions to their software. Problems with the system, though probably known during development, did not implement any safety interlocks to stop chemical mixings without at least one of the safety implementations in place. AECL also violated ethical standards, even going so far as continuing sales of Therac-25 units after multiple accounts of product over dosages. Iran Air Flight 655 was shot down because the developers of AEGIS forced reboots after an active FOF signal was sent to a plane. Because the system was not forced to reboot, this could constitute as an ethical conflict.

Aftermath Management

When a safety-critical system does fail, it then becomes the responsibility of the company to follow up and take charge of the problem. There are many ways to deal with the problem; one of them being a process called closed loop corrective action (CLCA).

CLCA

CLCA is a process developed by the International Organization for Standardization (ISO) standards foundation in order to make sure companies respond correctly to reports of hazardous situations. The usage of the CLCA process may either be triggered by ISO 9000 noncompliance or by concerns from the company itself. Depending on the severity of the problem, the system may even require a complete rewrite of the working build [6]. CLCA, also known as corrective and preventive action, is something that is planned in order to prevent an action from happening in the future. Closed loop just means the development process is closed until problems with the system are corrected. It is described in [6] as:

> The pattern of activities which traces the symptoms of a problem to its cause, produces solutions for preventing the recurrence of the problem, implements the changes and monitors that the changes have been successful.

There are a couple of steps to using CLCA:

1. Identify the problem.
2. Find the root cause of the problem.
3. Define procedures to correct.
4. Implement the new changes.
5. Follow up on the changes.

Results

CLCA processes could have helped in both the Bhopal and Therac-25 incidents. Bhopal could have recalled their entire tank mixing solutions after the incident and created measures that prevented the system from running without safety measures in place.

Therac-25 could have used some sort of CLCA to handle some of the concerns by medical professionals after the system killed three people. AECL denied responsibility at first because they could not reproduce the problem. After the first scare, AECL did not do extensive testing on each claim, stating that the system was completely safe [2].

Conclusion

By using hazard identification and analysis techniques, many of the problems that plagued Union Carbide, the Patriot Missile System, AEGIS, and Therac-25 could have been avoided. These techniques allow for systems engineers and software developers to find and remove hazards and risks before they turn into liabilities. Along with these techniques, CLCA allows for companies to efficiently deal with system failures in an ethical way. Using these preset strategies and learning from past mistakes, companies can avoid lawsuits and make products safer for consumers.◆

References

1. Kopec, D., and S. Tamang. "Failures in Complex Systems: Case Studies, Causes, and Possible Remedies." ACM SIGCSE Bulletin, June 2007, Vol. 39, Issue 2 <http://portal.acm.org/citation.cfm?id=1272905>.
2. Levenson, N.G., and C.S. Turner. "An Investigation of the Therac-25 Accidents." Computer July 1993. ACM Digital Library, Los Alamitos, CA <http://portal.acm.org/citation.cfm?id=161477.161479&coll=guide&dl=>.
3. Thayer, R.H., and M.J. Christensen. Software Engineering, Vol. 1 – The Development Process. 2nd ed. Hoboken, NJ: John Wiley & Sons, 2005.
4. Chambers, L. A Hazard Analysis of Human Factors in Safety-Critical Systems Engineering. Proc. of the 10th Australian Workshop on Safety-Critical Systems and Software. Apr. 2006. Australian Computer Society Sydney. Australia, 2006.
5. Piner, M.G. "Computer Society and ACM Approve Software Engineering Code of Ethics." Computer Oct. 1999.
6. Dodd. B. "Closed Loop Corrective Action." Online Training Materials. Spring 2002 <www.freequality.org/sites/www_freequality_org/documents/Training/Classes%20Spring%202002/Closed%20Loop%20Corrective%20Action.ppt>.

About the Author

Corey P. Cunha is a software developer and project manager for Savard Engineering (SE), an engineering firm focused on design, development, project management, and system analysis, where he designs and codes various systems built on PHP, ASP.NET, and SQL Server for social Web sites. Through SE, he has also built and maintained systems involving Global Positioning System tracking and plotting capabilities. Cunha currently has an internship working with GE Healthcare working in unit test management, test documentation, and enforcing coding standards, and is enrolled in the software engineering program at Champlain College in Burlington, Vermont.

Savard Engineering
75 Ethan Allen DR
Burlington, VT 05401
E-mail: corey.cunha@mymail.champlain.edu

STC

Systems & Software Technology Conference

SAVE THE DATE,
PLAN NOW TO ATTEND!

20-23 APRIL 2009
SALT LAKE CITY, UT

ABSTRACTS/PAPERS MAY BE SUBMITTED FOR CONSIDERATION FROM 25 AUG – 19 SEPT 2008

EXHIBITOR REGISTRATION OPENS 25 AUGUST 2008

WHO SHOULD ATTEND:

Acquisition Professionals
Program/Project Managers
Programmers
Systems Developers
Systems Engineers
Process Engineers
Quality and Test Engineers

Seven Words You Can Never Say on a Flailing Project

Stand-up comedy is a short-lived profession. It requires quick wit and impeccable timing to deliver a steady stream of laughs to a short-fused audience. Comics are at the mercy of the audience, deftly playing off their disposition and predilection. Stand-up comedy is definitely not for the faint of heart; even the best comics burn out or decay. Some, like Richard Pryor, Lenny Bruce, and John Belushi implode. Others, like Eddie Murphy, Roseanne Barr, and Tim Allen go stale. The smart ones transition to more secure jobs like sitcoms (Jerry Seinfeld) or movies (Steve Martin). Few have long, productive careers in stand-up.

One exception is George Carlin, who passed away June 22 at age 71. Carlin had a remarkably long career of 50-odd years in stand-up. Sure, he had dry spells as Mr. Conductor and the narrator of Thomas the Tank Engine; nevertheless, stand-up was his staple, spanning generations. His genius was making you think while you laugh; for example, "... if crime fighters fight crime and fire fighters fight fire, what do freedom fighters fight?"

Carlin transformed stand-up comedy by introducing a subversive approach aimed at the hypocrisy in our daily lives. He produced a rich body of work covering a diverse variety of issues, but sadly he will be remembered most for the seven words you can never say on television.

Project managers, especially software project managers, share similar pressures with stand-up comedians. Good software project management requires quick wit, impeccable timing, and consistent delivery to touchy customers. It is definitely not for the faint of heart or half-hearted.

Rather than tender yet another list of software best practices, I offer an elegy to projects in trouble with the seven words you can never say on a flailing project. You never get that list. It's typically learned by trial and error, but I'll save you the pain. The words are: grit, bliss, instruct, blunt, customer, milestone, and wits.

Let's start with GRIT. Never say grit on a flailing project unless you are referring to abrasive granules. Avoid the grit of indomitable spirit and firmness of character – Rooster Cogburn grit, true grit. True grit implies project leaders that possess the tenacity to inspire and the courage to hold accountable. Settle for cheerleaders, task masters, or bean counters – but never true grit.

Never associate BLISS with a flailing project. Bliss is an unusual word for any project. Carlin inquired, "If lawyers are disbarred and clergymen defrocked, doesn't it follow that electricians can be delighted?" You, in turn, may ask, do engineers actually like projects that offer a challenge and sense of accomplishment? Yes, indeed, your staff became engineers because they like to build and are blissful building things that captivate, fascinate, and intrigue. Nip that in the bud. Introduce a slow burn of inanity, monotony, and mental torpor to draw oxygen out of the project, avert any bliss, and leave the project in pure engineering hell.

Never utter the word INSTRUCT on a flailing project. Autodidactic managers and engineers agree: they don't need no stinkin' instruction. Managers believe instruction wastes time; time is money and money is short. Engineers see instruction as a waste of time, time that should be spent with thingamajigs, doohickeys, and doodads that intrigue the mind. If you instruct, you impart knowledge. Imparting knowledge leads to mutual understanding. Mutual understanding could lead to direction, purpose, objectives, common processes, and collaboration. The next thing you know, project destruct will become project construct because you dared to instruct. Keep your head in the sands of credulity.

On a flailing project, you can never be BLUNT. Blunt will only invite the four horsemen of flourishing projects: realism, directness, honesty, and frankness. Imagine the shock you would impose if requirements were realistic and clear, plans direct and uncomplicated, budgets honest and pragmatic, and project communication frank and straightforward. Imagine the hurt feelings and bruised egos. Imagine the nerve-racking decisions and arduous trade-offs. For the love of political correctness, you can't allow that to happen. Play it safe. Remain idealistic, apposite, and equivocal.

CUSTOMER? What customer? Never mention the customer on a flailing project. Involving the customer will complicate things – after all, they can't even congeal requirements? In today's society, customers should drive up, order, pick-up, and drive off. No questions asked. Do not be appeasers. Leave your customers out of the loop.

If you find yourself on a flailing project, never bring up the word MILESTONE. After all, a milestone is a stone, stones weigh you down and milestones weigh you down with accountability. Why gild the lily with measures and meetings? Who wants to know where your project really stands? Who needs course corrections? If you must have milestones, choose one and only one; and at all costs avoid the milestone's progeny – the inchstone.

Finally, never, under any circumstance, allude to WITS on a flailing project. Nothing subjugates a project faster than astute team members who perceive relationships between seemingly incongruous or disparate concepts, designs, and processes. Top talent is expensive, headstrong, and hard to control. Stop dreaming of a *deus ex machina*. Save money with desultory, nebbish, dilatory engineers. They will validate your best guesses and ensure your worst fears. Follow Carlin's advice, "Never underestimate the power of stupid people in large groups."

I apologize to managers in control and command of their software projects. This article offers you little. With time on your hands, maybe you can solve Carlin's greatest conundrum, "When someone asks you, 'a penny for your thoughts' and you put your two cents in, what happens to the other penny?"

—Gary A. Petersen
Arrowpoint Solutions, Inc.
gpetersen@arrowpoint.us

Can You BackTalk?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CrossTalk, we also accept articles for the BackTalk column. BackTalk articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BackTalk article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BackTalk article, visit our Web site at <www.stsc.hill.af.mil>.

Homeland Security

Software Assurance Program

Software is essential to enabling the nation's critical infrastructure. To ensure the integrity of that infrastructure, the software that controls and operates it must be reliable and secure.

Visit https://www.us-cert.gov/SwA to learn more about the software assurance program and how you can become more involved.

Security must be "built in" and supported throughout the lifecycle.

Visit https://BuildSecurityIn.us-cert.gov to learn more about the practices for developing and delivering software to provide the requisite assurance.

Sign up to become a free subscriber and receive notices of updates.

Software Assurance Working Groups

The Department of Homeland Security National Cyber Security Division has established stakeholder working groups to support each of its Software Assurance Program's focus areas. The goal of the working groups is to continue to support Software Assurance initiatives by bringing together members of government, industry, and academia with vested interests in software security to discuss and promulgate applicable practices and methodologies.

Software Assurance Focus Area and Working Group Matrix

https://buildsecurityin.us-cert.gov/swa/

The Department of Homeland Security provides the public-private framework for shifting the paradigm from "patch management" to "software assurance."

CROSSTALK / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CROSSTALK is co-sponsored by the following organizations: